Copyright Information

Published in the United States by Alta3 Research, Inc.

4 Bonnywick Drive
Harrisburg, PA 17111
P hone:*717-566-4428*
Web: *https://alta3.com*
E mail:*info@alta3.com*

Welcome!

Thank you for selecting Alta3 Research as your training provider! Since 1997, Alta3 Research has been empowering organizations and individuals using a no-nonsense approach to IT and DevOps training. We specialize in technologies such as Python, Ansible, 5G, Software Defined Networking, Microservices, Kubernetes, GoLang, Jenkins, GitHub, GitLab and more.

Alta3 Research excels at taking the advanced technologies required for IT and DevOps professionals, breaking them down and building proven training courses with over 95% student satisfaction. Training is available at the customer's site or virtually through instructor-led webinars. For more information on additional courses to help you achieve your career goals, please visit us at https://alta3.com

PDF Content Expiration

- **PDF created: 2023-02-14**

- **Refresh your content at: https://alta3.com**

Subscribe to us on these social media platforms for updates and free training!

- **YouTube:** https://youtube.com/alta3research
- **Twitter:** @alta3research
- **Facebook:** https://facebook.com/alta3research
- **LinkedIn:** https://www.linkedin.com/company/alta3-research-inc

Table of Contents

APIs and API Design with Python

1. Welcome to the Alta3 Research Lab Environment

Lab Objective

These labs will guide you through **hands-on, step-by-step exercises**, with *detailed explanations for each step*.

The virtual environment you will use to perform these labs is available to you **24/7** throughout the duration of your course. *We do not turn them off in between sessions*.

Getting Started

Let's make sure you are ready to start running through these labs.

Even so, you may have to use a new key-combination to use the copy-paste functionality. Let's practice it now.

If you click on the little clipboard icon in a lab, it will copy whatever text or code block is next to it, and will be available to paste into the virtual machine.

```
student@bchd:~$ # copy me!
```

After clicking on the , you will then be able to paste the command or code block wherever you would like.

Typically, you will want to paste into the Virtual Machine's terminal.

Each browser is different, but there are several ways to do this:

1. In the terminal, right click, then select `paste` (or) `paste as plain-text` (depending on the browser)
2. `Shift` `Insert`
3. `Ctrl` `Shift` `v`

Go ahead and try to paste the text of `# copy me!` into your terminal now.

Fundamental Commands

Not every person in this course has experience using a terminal to interact with a computer. This section is for those who may not have previous experience using a Linux terminal.

First of all, take a look at your command prompt to try to understand what it can teach us. It should look like:

```
student@bchd:~$
```

- **student** is the name of the user
- **bchd** is the hostname of the machine
- **~** (to the right of the colon) shows us the present working directory. Specifically, **~** refers to this user's home directory of `/home/student`.
- **$** shows us that this is a typical user (vs. **#** would indicate that it is the **root** user)

Now let's run some fundamental commands to get to know our environment a little bit better. Remember, we are starting in our `/home/student` directory for this.

1. **pwd** - [present working directory] shows you what directory you are in.

   ```
   student@bchd:~$ pwd

   /home/student
   ```

2. **ls** - list out the contents of the current directory.

   ```
   student@bchd:~$ ls

   static
   ```

3. **cd** - [change directory] - allows you to move to a different directory. Here we can move to the static directory.

   ```
   student@bchd:~$ cd static
   ```

4. **mkdir** - [make directory] - allows you to make a new directory. Let's make one called **training**.

```
student@bchd:~/static$ mkdir training
```

Also, let's make sure we can see that we made this **training** directory.

```
student@bchd:~/static$ ls

training
```

And let's move into the **training** directory.

```
student@bchd:~/static$ cd training
```

5. **touch** - makes a blank file.

```
student@bchd:~/static/training$ touch example_01.txt
```

Verify that the file named **example_01.txt** is there now.

```
student@bchd:~/static/training$ ls

example_01.txt
```

6. **echo** - returns text to the standard output.

```
student@bchd:~/static/training$ echo Alta3 Research Training rocks!

Alta3 Research Training rocks!
```

The > character allows us to redirect standard output to a different place, normally a file.

```
student@bchd:~/static/training$ echo Alta3 Research has AWESOME labs! > myfile.txt
```

The >> characters allow us to redirect standard output and append it to the end of a file.

```
student@bchd:~/static/training$ echo Alta3 Research has AMAZING labs! >> myfile.txt
```

7. **cat** - [concatenate] prints file(s) to standard output.

```
student@bchd:~/static/training$ cat myfile.txt

Alta3 Research has AWESOME labs!
Alta3 Research has AMAZING labs!
```

8. **mv** - [move] allows us to move a file or directory (often used to rename files).

```
student@bchd:~/static/training$ mv myfile.txt ego_fuel.txt
```

Verify that the file has moved.

```
student@bchd:~/static/training$ ls

ego_fuel.txt example_01.txt
```

Make sure that the contents of the file have not changed.

```
student@bchd:~/static/training$ cat ego_fuel.txt

Alta3 Research has AWESOME labs!
Alta3 Research has AMAZING labs!
```

9. **history** - shows all of the commands performed in this shell session.

```
student@bchd:~/static/training$ history
```

```
# copy me
pwd
ls
cd static
mkdir training
ls
cd training
touch example_01.txt
ls
echo Alta3 Research Training rocks!
echo Alta3 Research has AWESOME labs! > myfile.txt
echo Alta3 Research has AMAZING labs! >> myfile.txt
cat myfile.txt
mv myfile.txt ego_fuel.txt
ls
cat ego_fuel.txt
history
```

10. **man** - [manual] show the manual pages (aka documentation) for a given command. This is helpful for understanding any commands you may not feel comfortable with.

    `student@bchd:~/static/training$ man ls`

    To quit out of this view, press the keyboard key **q**

Excellent work! Now, let's move back to the home directory before you start working on the next lab.

`student@bchd:~/static/training$ cd ~`

Your prompt should now be back to: `student@bchd:~$`

If you have any questions throughout the course, please feel free to reach out to:

- Your Instructor
- Live Help via the Alta3 Research Discord Channel
- Email the Alta3 Research's Support Team support@alta3.com

Helpful Resources

Alta3 Research Instruction & Training

Alta3 Research Posters & Cheat Sheets

Alta3 Research YouTube

Common Questions and Solutions

- **My screen has "dots" around it :(**
  - *If you have more than one screen open, the "smallest" resolution wins. Therefore, the solution is to close the second tab you have open.*
- **I typed exit too many times and my CLI session closed!**
  - *Press the Refresh icon on your browser to refresh your session and create a new tmux session*
- **How do I get my content out of the tmux (CLI) environment?**
  - *The "best" way is probably by using git and GitHub. Alternatively, if you move content into the `~/static/` folder, you may access it by changing the "source" of your right-most pane in the split-screen session. To change the source, click on the icon that looks like three sheets of paper in the upper-right hand corner.*
- **Will the lab environments "shut off" this week?**
  - *No. The your lab environment is on 24x7, until the termination date.*
- **Can I close the tab to live.alta3.com**
  - *Yes! Unless you clear your internet cache, you should just be able to revisit your custom course link to regain access. If you are asked to log-in again, simply use the same email address and handle to regain access.*

2. Python 201 - APIs and API Design with Python

- **5 Day Course**
- **Lecture and Lab**
- **Every course includes the opportunity to earn an API Design with Python certification from Alta3 Research.**

Course Overview

Application Programming Interfaces (APIs) have become increasingly important as they provide developers with connectivity to everything from rich datasets in an array of formats (such as JSON) to exposing the configurability of software applications and network appliances. Lessons and labs focus on using Python to interact, design, and build APIs for the purposes of scripting automated solutions to complex tasks. Class is a combination of live demonstrations and hands-on labs.

What You'll Learn

- Client side Python Scripting to RESTful (and non-RESTful) APIs
- Design RESTful API interfaces with Flask Web Framework
- Overview of Django
- Deploy your Python web apps as Docker containers
- Parse and manipulate popular data structures (JSON, CSV, Excel, and YAML) as pandas dataframes
- Best practice techniques

Course Outline:

1. **Python Review**
    - Version Control with Git
    - Lists
    - Dictionaries
    - Conditionals (if, elif, else)
    - Loops (for and when)
    - Functions
    - Classes and Methods
    - Using pip
2. **Working with Data - JSON, YAML, CSV and Excel**
    - JSON RFC 7159
    - JSON Formatting
    - YAML intro
    - YAML Formatting
    - Python Libraries for decoding JSON, YAML and CSV
    - Reading and Writing to Excel
    - Dataframes and pandas
3. **Web and RESTful APIs**
    - Creating an HTTP Client & Server with Python
    - Introduction to REST
    - RESTful API on Etcd keystore (Kubernetes distributed DB)

- Creating a Python client to interact with API endpoints
- Authentication
- API dev keys
- Secure password retrieval
- Tokens and APIs
- OAuth v2.0

4. **API Design Practices**

- RESTful Architecture
- SOAP overview
- Stubbing code with Swagger
- Describing Resource functionality (GET, POST, PUT, DELETE, etc.)
- Collections, resources, and URLs
- Using nouns, not verbs
- Understanding HTTP status codes
- Returning data

5. **Building APIs with Flask**

- Flask Overview
- Decorators
- Building APIs with Python and Flask
- APIs returning Jinja2 templating
- Returning a 'cookie'
- Building Sessions
- Redirecting from URIs
- Build an API to accept a file upload
- Overview of Django

6. **Database Integration**

- Overview
- Connecting to Python
- Read / Write operations
- Other useful instructions
- Connecting APIs and SQLite
- Python and PostgreSQL
- Python and MongoDB
- Reading and Writing to Databases with APIs

7. **Deploying APIs within Enterprise**

- Docker containers
- Docker build
- Constructing Docker images
- Dockerfile
- Deploying a Flask App on Docker
- Automating build processes

8. **Processes and Threads**

- Threading
- Concurrency
- Async.io
- Context change
- Deadlock errors
- Thread starvation
- Racing conditions and racing specifics
- Working with Locks

Hands On Labs:

1. Welcome to Alta3 Research Labs
2. Using vim
3. Introduction to VScode
4. Tmux Basics
5. SCM with GitLab
6. API Design with Python - Certification Project

7. LECTURE - Introduction to APIs
8. LECTURE - Object Oriented Programming for APIs
9. Getting dir(obj) help() and pydoc
10. LECTURE - Practical Application of Lists
11. Lists
12. LECTURE - Practical Application of Dict
13. Dictionaries
14. CHALLENGE - List and Dict Modeling
15. Your First API Request
16. LECTURE - Python data types vs JSON
17. Python Data to JSON file
18. pandas dataframes - MS Excel, csv, json, HTML and beyond
19. LECTURE - Introduction to HTTP
20. Standard vs. Third Party Libraries and Open APIs
21. requests library - Open APIs
22. requests library - RESTful GET and JSON parsing
23. LECTURE - Visualizing API response data with Py
24. CHALLENGE - Key-pairs and HTTP GET
25. LECTURE - HTTP GET vs POST
26. requests library - GET vs POST to REST APIs
27. APIs and Dev Keys
28. RESTful APIs and Dev Keys
29. LECTURE - OAuth
30. SOAP and Python
31. Construct a SimpleHTTPServer and HTTP Client
32. LECTURE - Intro to Flask
33. Building APIs with Python
34. LECTURE - Introduction to Jinja
35. Flask APIs and Jinja
36. CHALLENGE - Jinja
37. CHALLENGE Solution
38. Flask APIs and Cookies
39. Flask Sessions
40. LECTURE - Controlling your APIs
41. Flask Redirection, Errors, and API Limiting
42. Flask Uploading and Downloading Files
43. LECTURE - Learning sqlite3
44. Tracking API Data with sqlite3
45. Tracking Inventory with sqlite3
46. Building a Production Server
47. Running Flask in a Docker Container
48. LECTURE - etcd REST API
49. etcd and RESTful Client-side Design
50. Logging API Behavior
51. Swagger
52. Designing Our Own API
53. Introduction to Django
54. LECTURE - Introduction to Threads
55. Working With Threads
56. Threading API requests
57. Introduction to Async IO
58. Glossary

Certification:

- Alta3 Research Python 201 - API and RESTful API - Certification Project

Prerequisites:

- Recommended Prerequisite: Python Basics (5 days)
- Coding experience in another language serves as an adequate prerequisite

Who Should Attend:

- System Administrators
- Network Engineers
- Software Developers
- Python Enthusiasts

Follow-on Courses:

- Python 202 - Python for Network Automation (5 days)
- Jenkins Automation Server Essentials (2 days)
- Git and GitHub (or Git and GitLab) (2 days)
- Terraform 101 - Infrastructure as Code (3 days)

3. Using Vi and Vim

Throughout the course, you'll find our documentation suggests using the vim text editor. Vim is an improved version of vi, so if you know vi, you'll just be refreshing some basic skills in this lab.

Vim is the editor of choice for many developers and power users. It's a "modal" text editor based on the vi editor written by Bill Joy in the 1970s for a version of UNIX. It inherits the key bindings of vi, but also adds a great deal of functionality that is missing from the original vi.

In most text editors, the alphanumeric keys are only used to input those characters unless they're modified by a control key. In vim, the mode that the editor is in determines whether the alphanumeric keys will input those characters or move the cursor through the document. This is what is meant by 'modal.' When you first enter vim, you enter in the command mode.

Procedure - Using Vim

1. Review ***(read-only)*** the following **vim** commands:
    - **To start editing changes by entering the --INSERT-- mode:**
        - press `i`
    - **To stop editing and return to command mode:**
        - press `ESC`
    - **To save and quit:**
        - press `SHIFT` + `:` *press SHIFT and the COLON keys at the same time*
        - type `wq` (write out and quit)
        - press `ENTER` to confirm
    - **To quit without saving:**
        - press `SHIFT` + `:` *press SHIFT and the COLON keys at the same time*
        - type `q!` (quit and ignore all changes)
        - press `ENTER` to confirm
2. Move to the student home directory.

```
student@bchd:~$ cd
```

3. Now create a text file within the vim environment.

```
student@bchd:~$ vim zork.test
```

4. Vim is entered in command mode. To write text, you'll need to change to **--INSERT--** mode. To begin writing text, press:
    - `i`
5. Notice in the bottom left corner of the screen it now says **--INSERT--**
6. Type a few sentences. Be sure to include some carriage returns, like the following:

```
West of House
You are standing in an open field west of a white house, with a boarded front door.
There is a small mailbox here.
```

7. Okay, great! Now leave **--INSERT--** mode, and return to command mode, by pressing the escape key.
    - `Esc`
8. Notice that **--INSERT--** no longer is at the bottom left of the screen. Generally, pressing the Escape key will always return you to the command mode.
9. Use the directional arrow keys on the keyboard to move the cursor around the screen.
10. Perform the following to save changes, and return to the command line.
    - press `SHIFT` + `:`

press SHIFT and then the COLON key at the same time

- type `wq` (write out and quit)
- press `ENTER` to confirm

11. Confirm the file saved correctly by printing its contents. We can use the **cat** command to catenate, or read data from files and print their contents to the screen.

```
student@bchd:~$ cat zork.test
```

12. Edit the file zork.test again.

```
student@bchd:~$ vim zork.test
```

13. Remember, you enter vim in command mode. Take advantage of that and press the following capital letter to jump to the end of the file:

    - press `SHIFT` + `g`

press SHIFT and the g keys at the same time

14. Press the following capital letter to begin appending at the end of the line (enter --INSERT-- mode at the end).

    - press `SHIFT` + `a`

press SHIFT and the a keys at the same time

15. You'll notice it says **--INSERT--** at the bottom left of your screen again. Once again you can type normally. Add additional text, such as the following:

```
West of House
You are standing in an open field west of a white house, with a boarded front door.
There is a small mailbox here.
Open mailbox
Opening the small mailbox reveals a leaflet.
Read leaflet
(leaflet taken)
"WELCOME TO ALTA3 LABS!"
```

16. Okay, great! Now leave **--INSERT--** mode, and return to command mode by pressing the escape key.

    - `Esc`

17. Perform the following to return to the command line **without** saving any changes.

    - press `SHIFT` + `:`

press SHIFT and the COLON keys at the same time

- type `q!` (quit without saving)
- press `ENTER` to confirm

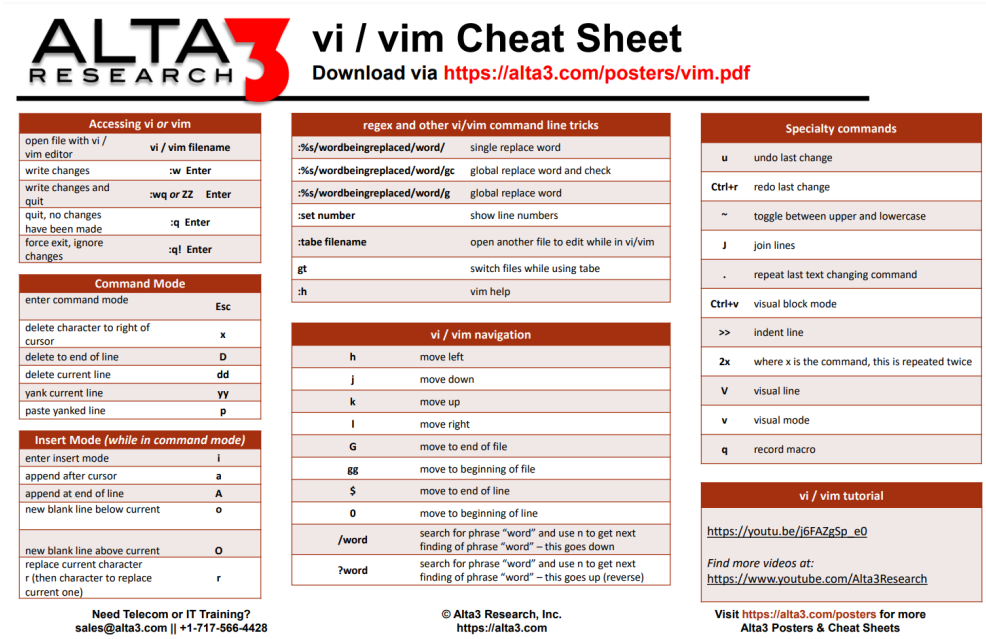
18. Confirm that none of the changes you just made were saved.

```
student@bchd:~$ cat zork.test
```

19. Remove the file.

```
student@bchd:~$ rm zork.test
```

20. Review a vim cheat sheet. These make very useful wall art and you may have seen one hanging in a colleague's workspace.



vi / vim Cheat Sheet

Download via https://alta3.com/posters/vim.pdf

Accessing vi *or* vim	
open file with vi / vim editor	vi / vim filename
write changes	:w Enter
write changes and quit	:wq *or* ZZ Enter
quit, no changes have been made	:q Enter
force exit, ignore changes	:q! Enter

Command Mode	
enter command mode	Esc
delete character to right of cursor	x
delete to end of line	D
delete current line	dd
yank current line	yy
paste yanked line	p

Insert Mode *(while in command mode)*	
enter insert mode	i
append after cursor	a
append at end of line	A
new blank line below current	o
new blank line above current	O
replace current character r (then character to replace current one)	r

Need Telecom or IT Training?
sales@alta3.com || +1-717-566-4428

regex and other vi/vim command line tricks	
:%s/wordbeingreplaced/word/	single replace word
:%s/wordbeingreplaced/word/gc	global replace word and check
:%s/wordbeingreplaced/word/g	global replace word
:set number	show line numbers
:tabe filename	open another file to edit while in vi/vim
gt	switch files while using tabe
:h	vim help

vi / vim navigation	
h	move left
j	move down
k	move up
l	move right
G	move to end of file
gg	move to beginning of file
$	move to end of line
0	move to beginning of line
/word	search for phrase "word" and use n to get next finding of phrase "word" – this goes down
?word	search for phrase "word" and use n to get next finding of phrase "word" – this goes up (reverse)

© Alta3 Research, Inc.
https://alta3.com

Specialty commands	
u	undo last change
Ctrl+r	redo last change
~	toggle between upper and lowercase
J	join lines
.	repeat last text changing command
Ctrl+v	visual block mode
>>	indent line
2x	where x is the command, this is repeated twice
V	visual line
v	visual mode
q	record macro

vi / vim tutorial
https://youtu.be/j6FAZgSp_e0
Find more videos at: https://www.youtube.com/Alta3Research

Visit https://alta3.com/posters for more Alta3 Posters & Cheat Sheets

Get all of Alta3's Cheat Sheets here! https://alta3.com/posters

21. Don't worry if you got stuck a few times, go back and try it again! Working within the vim environment is a basic Linux admin skill that is useful to everyone that expects to work at the Linux CLI.

4. Using VSCode

While vim is a great place to start learning how to code within a terminal, many programmers prefer to work in VSCode due to its easy-to-navigate integration with git and GitHub, debugging features, graphical user interface (GUI), and more. Plus, once you are comfortable with the keyboard commands in vim, VSCode actually has an option to integrate those key bindings from vim, so there's no additional shortcuts you need to learn!

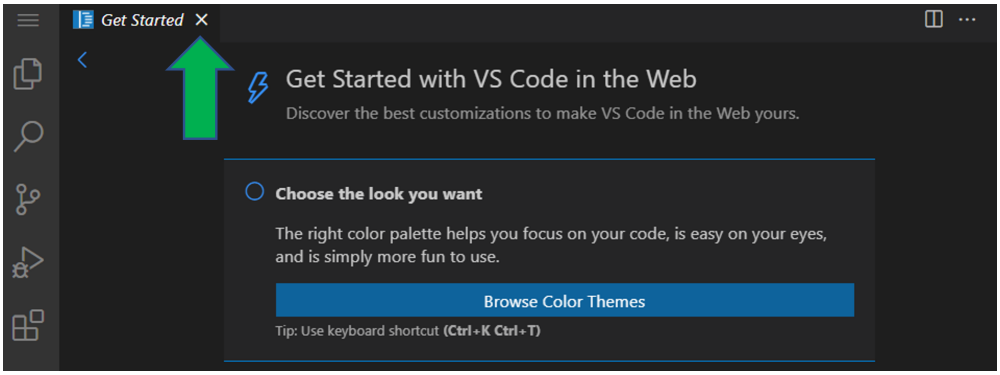
VSCode User interface documentation

Lab Objective:

The objective of this lab is to learn how to enter and navigate VSCode within your lab environment, configure VSCode to use vim shortcuts, and create a text file similar to the previous lab in vim. You will also be introduced to the concepts of a GitHub repository and writing a python script in VSCode, but we will revisit those topics more in-depth in later labs.

Procedure:

1. Enter VSCode by clicking on the icon that looks like three sheets of paper located at the top right of your tmux terminal.
2. In the drop-down menu, select the option `code`.
3. You are now in VSCode! That was easy, right? You'll be greeted by a "Getting Started" tab which we absolutely don't need. Go ahead and click the x to close it.



4. Let's tour some of the VSCode interface. Click on the three horizontal lines. This is the ***application menu*** where you can create and open new files, and set up your virtual workspace to your liking. For now, just look through the various tools in this menu.

5. Next, click on the icon below with the two sheets of paper. This is the ***explorer*** which features a more concise "Open Folder" option than the application menu, as well as an option to "Clone Repository". This is a git/GitHub term, which allows you to either work individually or collaborate on projects. More on that later.

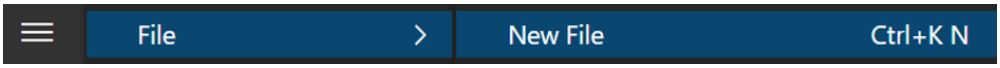
6. Below the explorer is a magnifying glass icon which, you guessed it, is a ***search*** tool. Here you can easily find and replace words or phrases in a given file.

7. Next is a little roadmap-looking icon. This is your ***source control*** feature, where you will be able to see changes and make GitHub commits. Again, don't worry about these terms yet.

8. Second to last is the highly useful ***run and debug***, which looks like a play button and some kind of insect (get it?). This feature allows you to view and fix any issues that may exist within your code.

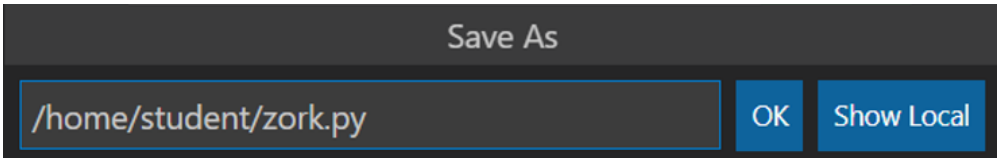
9. Finally, the building-blocks icon is where you can install ***extensions*** to your VSCode workspace.

10. In VSCode, press the ≡ button in the upper left. Select `File` > `New File`.



11. Press `ctrl s`, or go back to the ≡ button in the upper left. Select `File` > `Save`. When prompted, save your file as `/home/student/zork.py`.

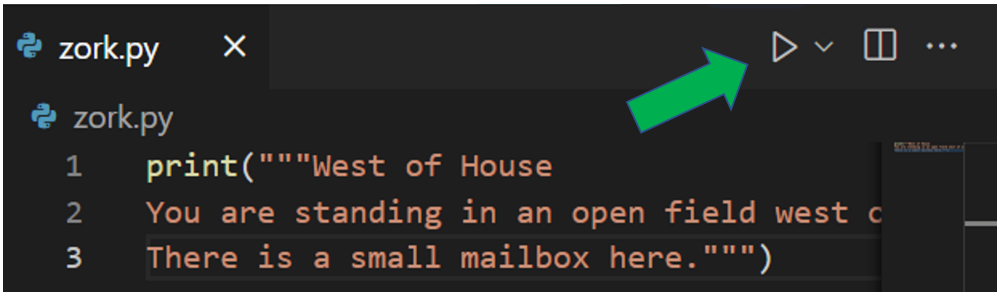
> The `.py` extension is important! VSCode will recognize your file as a Python script and will format it appropriately.

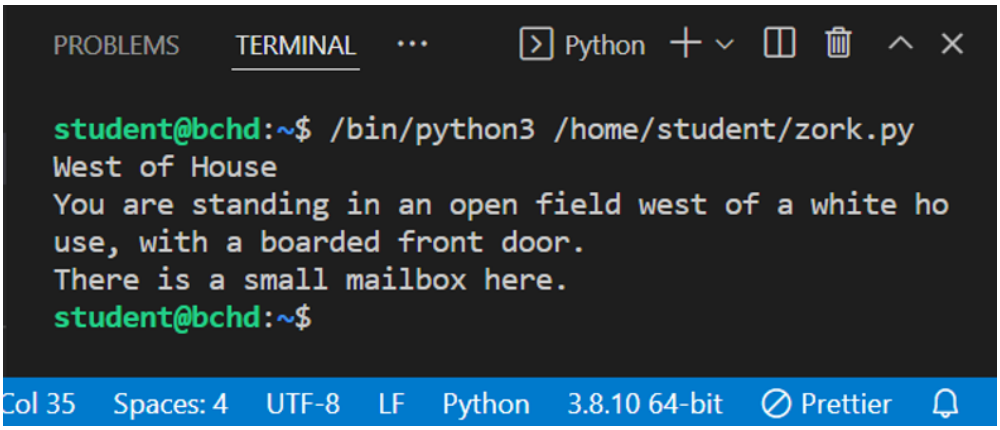


12. Copy the following by clicking the clipboard icon in the upper right of the text field below, then paste it into your file:

```
print("""West of House
You are standing in an open field west of a white house, with a boarded front door.
There is a small mailbox here.""")
```

13. In the upper right of your VSCode window, click the play ▶ button. This will open a terminal window at the bottom of your screen where your script will be called and executed!





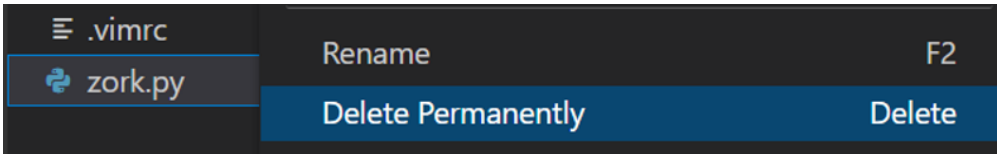
14. You can also use the terminal as a normal Linux command line. Display that code you just wrote:

```
student@bchd:~$ cat ~/zork.py
```

15. You can also call your Python modules from the command line as well. Type in the following:

```
student@bchd:~$ python3 ~/zork.py
```

16. No need to keep this file. In the "Explorer" menu on the left side of your VSCode screen, right-click `zork.py` and select "Delete Permanently".



17. There is a whole lot more functionality to explore within VSCode, but those are the basics to get you started. Feel free to explore!

5. Tmux

The program known as **tmux** is a very powerful tool for boosting the efficiency of using the command line. It allows you to *multiplex* the *terminal*, thus enabling you to effectively multiply the speed of your command line usage.

Learning Objective(s):

- Become familiar using **tmux** to massively increase your CLI efficiency

Procedure - Tmux

1. This lab is about learning to use tmux. You are already in a tmux session. If you weren't, you could apt-install **tmux**, then type the command `tmux` to get started. But the first thing we need to practice is a two-step process to enter into the command mode of **tmux**. Follow these steps:

    1. Hold the **Ctrl** key
    2. Tap the **B** key
    3. Take your hands off the keyboard (alias HOK)

    > Practice doing this seven times.

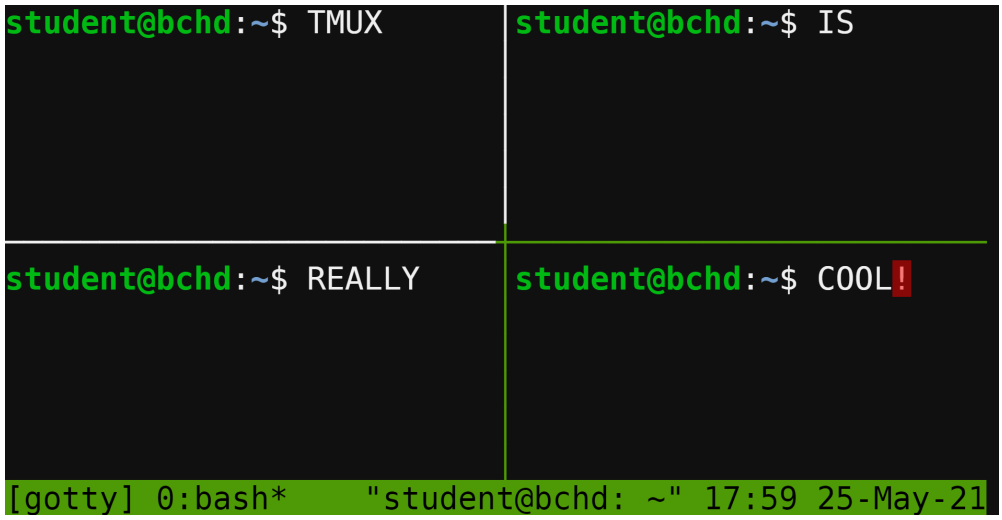
2. Now you are ready to learn some of the most essential **tmux** commands.

    For each of the following characters, enter the **tmux** command sequence (**Ctrl B**, HOK), then push the character. Note, not all of the characters are single-button presses. For example: `%` can be done by hitting the keyboard buttons `Shift` and `5`.

    - **%** - Vertical Split
    - **"** - Horizontal Split
    - **z** - Zoom/Unzoom
    - **←** - Move cursor leftwards one pane
    - **↑** - Move cursor upwards one pane
    - **→** - Move cursor rightwards one pane
    - **↓** - Move cursor downwards one pane
    - **[** - Scrollback (type `q` to exit)
    - **?** - Show the list of available tmux commands (type `q` to exit)

3. To exit out of a **tmux** pane, you can simply type the word `exit`, `logout`, or hit the key sequence **Ctrl D**.

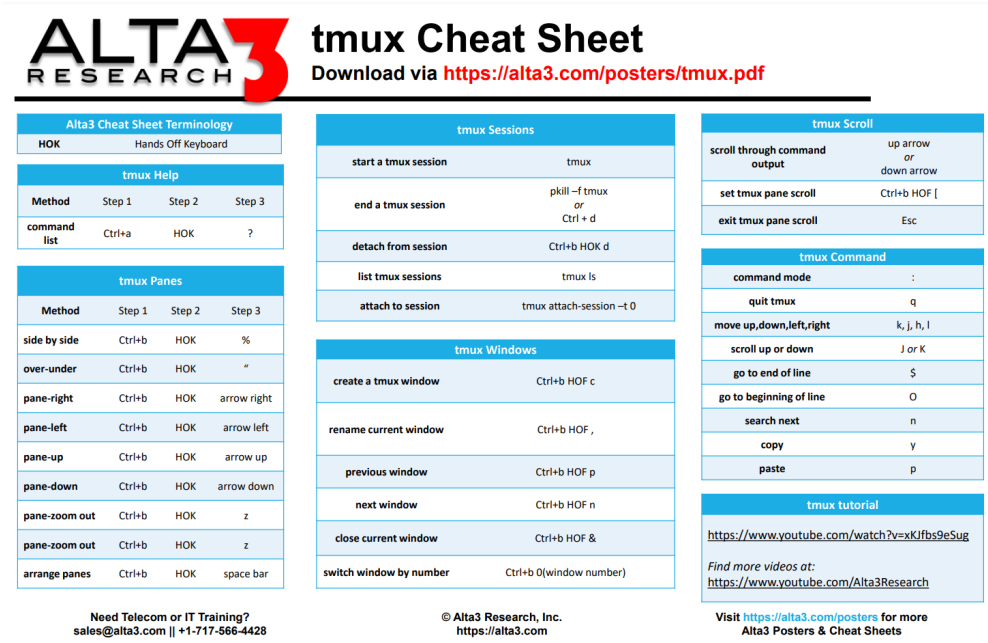
4. Challenge: Make your screen look like this:



    If you are struggling to do this, please re-watch the video or ask your instructor for help.

5. After completing this challenge, eliminate all but one of the panes before moving on to the next lab.

6. For more tmux commands, check out our tmux cheat sheet:



tmux Cheat Sheet

Download via https://alta3.com/posters/tmux.pdf

Alta3 Cheat Sheet Terminology	
HOK	Hands Off Keyboard

tmux Help			
Method	Step 1	Step 2	Step 3
command list	Ctrl+a	HOK	?

tmux Panes			
Method	Step 1	Step 2	Step 3
side by side	Ctrl+b	HOK	%
over-under	Ctrl+b	HOK	"
pane-right	Ctrl+b	HOK	arrow right
pane-left	Ctrl+b	HOK	arrow left
pane-up	Ctrl+b	HOK	arrow up
pane-down	Ctrl+b	HOK	arrow down
pane-zoom out	Ctrl+b	HOK	z
pane-zoom out	Ctrl+b	HOK	z
arrange panes	Ctrl+b	HOK	space bar

Need Telecom or IT Training?
sales@alta3.com || +1-717-566-4428

tmux Sessions	
start a tmux session	tmux
end a tmux session	pkill –f tmux *or* Ctrl + d
detach from session	Ctrl+b HOK d
list tmux sessions	tmux ls
attach to session	tmux attach-session –t 0

tmux Windows	
create a tmux window	Ctrl+b HOF c
rename current window	Ctrl+b HOF ,
previous window	Ctrl+b HOF p
next window	Ctrl+b HOF n
close current window	Ctrl+b HOF &
switch window by number	Ctrl+b 0(window number)

© Alta3 Research, Inc.
https://alta3.com

tmux Scroll	
scroll through command output	up arrow *or* down arrow
set tmux pane scroll	Ctrl+b HOF [
exit tmux pane scroll	Esc

tmux Command	
command mode	:
quit tmux	q
move up,down,left,right	k, j, h, l
scroll up or down	J *or* K
go to end of line	$
go to beginning of line	0
search next	n
copy	y
paste	p

tmux tutorial

https://www.youtube.com/watch?v=xKJfbs9eSug

Find more videos at:
https://www.youtube.com/Alta3Research

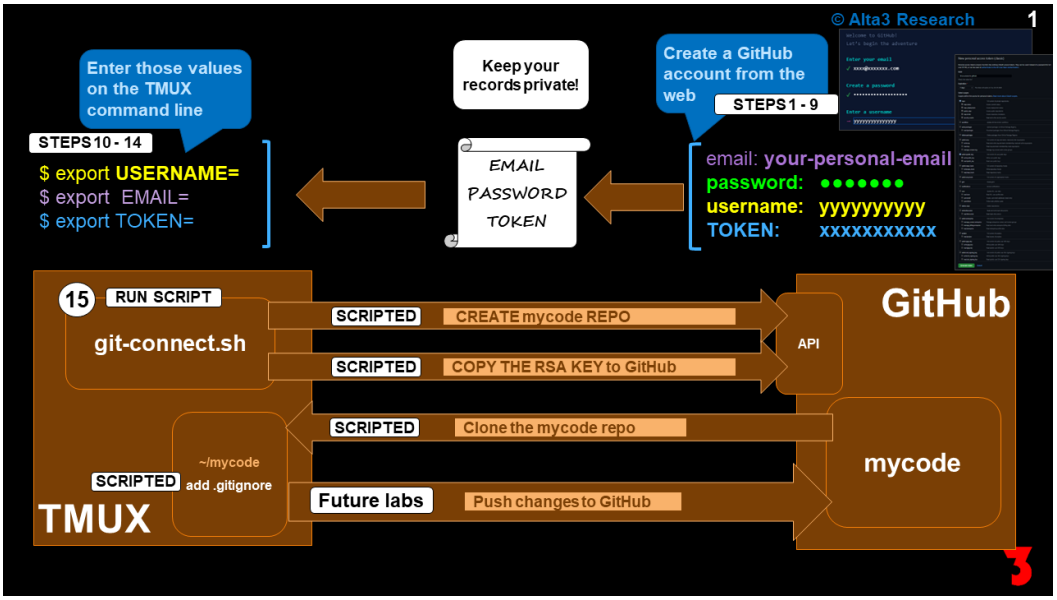
Visit https://alta3.com/posters for more Alta3 Posters & Cheat Sheets

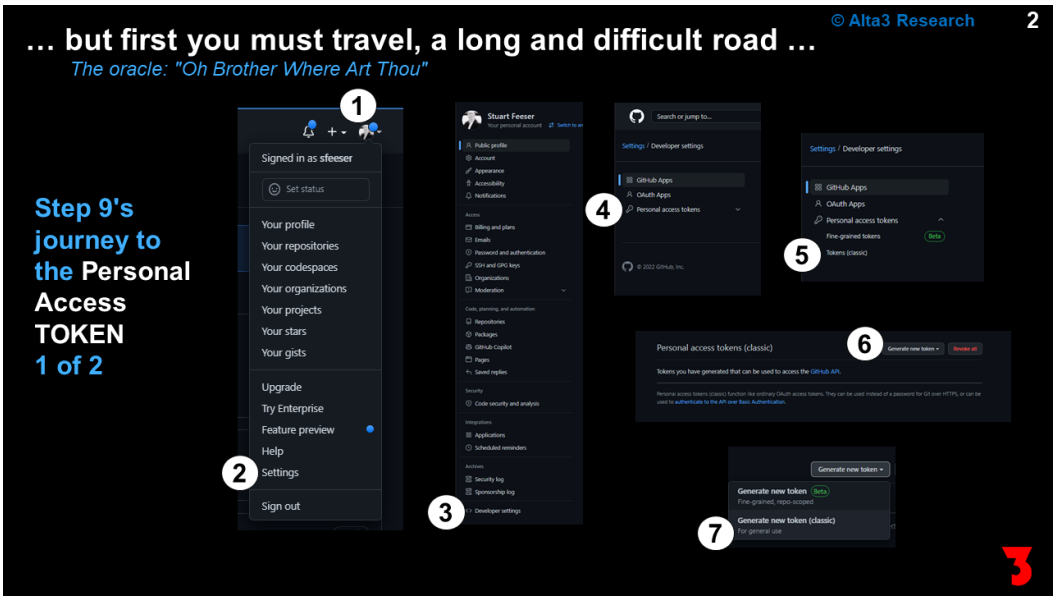
Get all of Alta3's Cheat Sheets here! https://alta3.com/posters

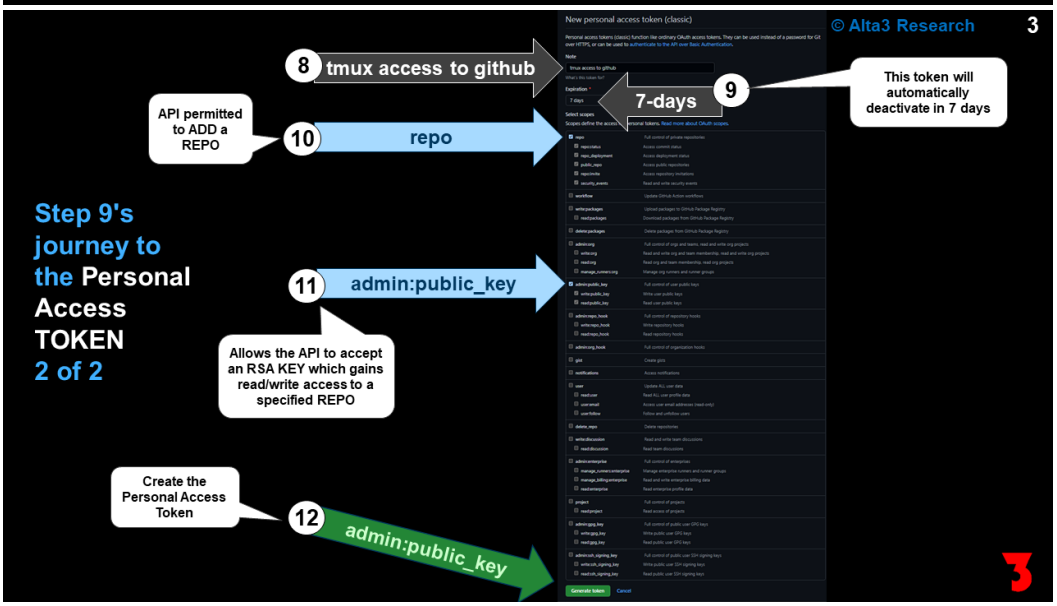
6. Revision Control with Git and GitHub

In this lab, we're going to explore SCM, or Software Control Management platforms. Git is the de-facto tool for tracking and version controlling your work. Git is a tool that we run on our local machine. GitHub is an HTTPS browser-friendly platform that syncs with git, and makes your code available to the world. This is a good thing.

In this lab, you'll make a GitHub account. This is free and will allow you to save the code you develop this week.







Section 1 - Create a GitHub account

1. In your browser, open a new tab and navigate to https://github.com

2. Click on `Sign-up` on the landing page.
3. Enter the following information.
    - **Email Address** - Use an email address you check often.
    - **Password** - Always make passwords unique and ultimately change them often.
    - **Username** - This handle will be shared with career professionals.
4. STOP: before you go any farther, record the above information. You will need it later.
5. Now click on the **Create account** button at the bottom of the screen.
6. At the bottom of the screen, click the green `Continue` button.
7. Answer the questions as best you can to help the GitHub metrics, and then click `Submit` at the bottom of the screen.
8. You'll need to verify your email address. Check the email address you used to sign up, and click on the link or button they sent to you.
9. Create your Personal Access Token. Complete the following tasks:

1 Click on the `tiny round icon in the upper right corner`
2 Click `Settings` on the dropdown
3 Click `Developer settings` on the bottom of the left menu
4 Click `Personal access tokens`
5 Click `Tokens (classic)`
6 Click `Generate new token`
7 Click `Generate new token (classic)`
8 Title the key to "tmux access to github"
9 Check the box for `repo`
10 Check the box for `admin:public_key`
11 Click the green `Generate token` button at the bottom
12 AT THE TOP OF THE PAGE, COPY THE NEWLY GENERATED TOKEN TO YOUR CLIPBOARD

Section 2 - Back on TMUX, connect to your new githib account

10. Save your **TOKEN**. Replace XXXXXXXXX with the token you just copied from GITHUB.

```
student@bchd:~$  export TOKEN=XXXXXXXXX
```

11. Save your **USERNAME**. Replace XXXXXXXXX with your GITHUB username.

```
student@bchd:~$  export USERNAME=XXXXXXXXX
```

12. Save your **EMAIL**. Replace XXXXXXXXX with the EMAIL you used to create your github account.

```
student@bchd:~$  export EMAIL=XXXXXXXXX
```

13. Download the following bash script.

```
student@bchd:~$ wget https://labs.alta3.com/courses/github/scripts/git-connect.sh
```

14. Run the script.

```
student@bchd:~$ bash ~/git-connect.sh
```

Section 3 - Saving Your Work

15. The following steps are ones you should memorize. You'll be issuing these commands all the time. First issue `git status`, which will reveal if you added something and forgot about it. First move into your git directory you just made.

```
student@bchd:~$ cd ~/mycode
```

16. Create a file that we can commit.

```
student@bchd:~/mycode$ echo "Training and learning about git commits with Alta3 Research" > ~/mycode/Alta3Research.txt
```

17. Now issue `git status`, which will list all changes you've made since you last backed up your files to GitHub.

```
student@bchd:~/mycode$ git status
```

18. Next we'll describe what we want to add to our repo. We'll wildcard this, so everything in the `~/mycode/` directory is added.

```
student@bchd:~/mycode$ git add *
```

19. Time to perform a commit. This makes a new version.

```
student@bchd:~/mycode$ git commit -m "First commit to learn about version controlling"
```

20. Now push your new version to GitHub for tracking. By default, the keyword **origin** points to the repo you cloned the repo from.

```
student@bchd:~/mycode$ git push origin HEAD
```

21. You should get in the habit of issuing the following commands each time you have a success, or end for the day:

    - `git status`
    - `git add /home/student/mycode/*`
    - `git commit -m "reason for commit"`
    - `git push origin HEAD`

22. Great! Move back to your home directory.

```
student@bchd:~/mycode$ cd /home/student
```

Troubleshooting

1. Run this command to check the origin:

```
student@bchd:~/mycode$ git remote show origin | grep URL
```

   This is BAD (Using https will prompt you for login and password)

```
Fetch URL: https://github.com/xxxxxxx/mycode.git (The xxxxxxx must be your repo name)
Push  URL: https://github.com/xxxxxxx/mycode.git
```

   This is GOOD (SSH):

```
Fetch URL: git@github.com:xxxxxxx/mycode.git  (The xxxxxxx must be your repo name)
Push  URL: git@github.com:xxxxxxx/mycode.git
```

2. Correct the Fetch and Push URL with the following command. Replace the xxxxxxx with your repo name:

```
student@bchd:~/mycode$ git remote set-url origin git@github.com:xxxxxxxxxxx/mycode.git
```

3. Check your git config. An good example is show below:

```
student@bchd:~/mycode$ cat ~/mycode/.git/config
```

   Working example:

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = git@github.com:xxxxxx/mycode.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
```

4. Check if your github keys exist

```
student@bchd:~/mycode$ test -f ~/.ssh/id_rsa_github -a -f ~/.ssh/id_rsa_github.pub  && echo GOOD! github keys exists || echo NO KEYS!
```

> If the keys are missing, add them now.

> Alternatively, you can update the entry in the file `~/.ssh/config`. There is an entry for `github.com` that indicates the name of the private key to use (private keys *do not* have `.pub` extension). Change the entry the config file to reflect the name of the keypair that *you* created. Save and quit, then try to push to GitHub again.

Histories cannot be merged

If you are seeing this error, the safest and easiest solution is to delete the repository on GitHub.com (but not locally). Create a new mycode repo on GitHub *without* a `README.md` (or any other file). This is called a "bare" repository. Try pushing your code to this new repo.

Nothing to push

You need a commit to push to GitHub.com. Get a list of all the commits in your local repository with `git log`. If you don't see anything recent, you might need to perform the `git add` / `git commit` commands before you can get a commit to push to GitHub.

Other Useful Resources

- If you ever run into a problem using git / GitHub, let the instructor know. It is critical to start learning to version control code. You might want to spend some time clicking around on the following guides. They're quite short, and although you might not understand what they're talking about, they'll begin exposing you to the way we 'speak' when using git & version controlling software. Some getting started guides relating to GitHub can be found here: https://guides.github.com/
- Windows and Mac users might also check out the local client GitHub Desktop, available at https://desktop.github.com/ This client makes it **easy** to work through git via a local GUI on your Windows or Mac desktop environment.

7. SCM Option #2- GitLab

8. API Design with Python - Alta3 Research Certification Project

Lab Objective

The objective of this lab is to offer students an opportunity to obtain a **Python and RESTful API Certification** from Alta3 Research. This certification should work as proof of basic proficiency of Python and RESTful APIs. To earn a certificate, complete the following tasks. Code will be graded on a pass / fail basis.

- Certification submissions may be made at anytime.
- If you find your environment shutdown, and want a new environment, email **support@alta3.com**. Include the course name of the course you took so the proper corresponding environment may be created.
- How to submit is described below.

Procedure

1. Create a public repository on GitLab called **alta3research-mycode-cert**
2. Inside of the public repository should be (at least) **two (2)** scripts. One called, **alta3research-flask01.py** and a second called **alta3research-requests02.py**
3. Your script **alta3research-flask01.py** should demonstrate proficiency with the `flask` library. Ensure your application has at least two endpoints. At least one of your endpoints should return legal JSON.
4. Your script **alta3research-requests02.py** should demonstrate proficiency with the `requests` HTTP library. The API you target is up to you, but be sure any data that is returned is "normalized" into a format that is easy for users to understand.
5. It should be clear how to run your code. If there are any special steps or prerequisites, be sure they are explained in within the script and the `README.md` within your GitLab repository.
6. Your code should run without error when executed with Python 3.x
7. Be sure to observe best practice when writing both scripts. Some things to think about:
    - Include a shebang
    - Include documentation at the top
    - Comment your code so it is clear
    - Use functions
    - Your code made span more than a single file
    - A tool such as `pylint` may help you write better code
8. When you finish, email to **info@alta3.com** containing the following:
    - **Subject:** - `Alta3 Research Python Certification - Grading Request`
    - In the body be sure to include:
        - **Name:** - What you'd like printed on the certification
        - **Email:** - Email address you used to enroll in the course
        - **GitLab URL:** - The URL of your public GitLab repository containing your work to be graded
        - **Course Start:** - Date your course started on
        - **Instructor:** - Name of your instructor (if you remember)

9. Lecture - Intro to APIs

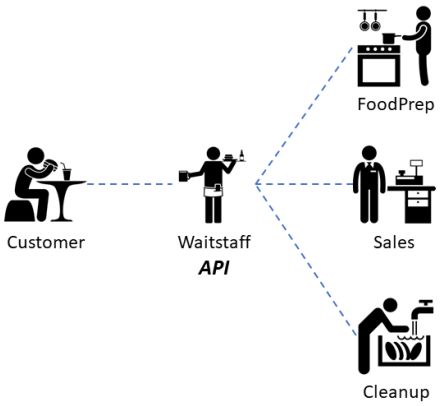
Lab Objective

The objective of this lab is to introduce the concepts governing Application Programming Interfaces (APIs).

Procedures

Introduction to APIs **© Alta3 Research**

Application Programming Interface (API)

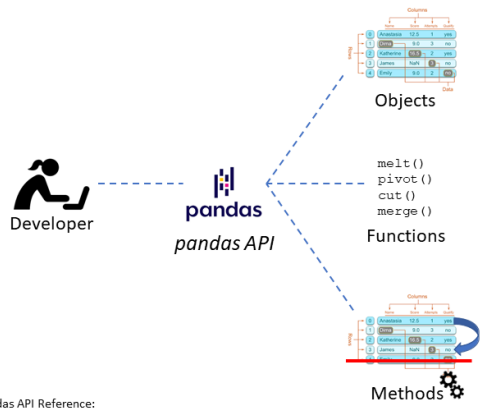


- APIs allow code to be accessed and executed
- Waitstaff (API) abstractions
  - Understands access to various services (software)
    - Food orders must be on a single ticket, legible, and audibly announced
    - Cash sales must be made at the downstairs register only
    - Never put sharps in a soapy sink
    - Location of clean utensils & dishes
  - Highly available and common interface
  - Prepped to anticipate changes
    - Ice Cream machine is broken
    - Accept cryptocurrency as valid payment
  - Security
    - Upset customer does not effect back of the house
    - Secret recipes are obfuscated (public vs private)
- APIs make delivery of services flexible

Introduction to APIs **© Alta3 Research**

Software APIs

Using pandas as an example, the API is the collection of objects, functions, and methods that expose interaction with pandas code.



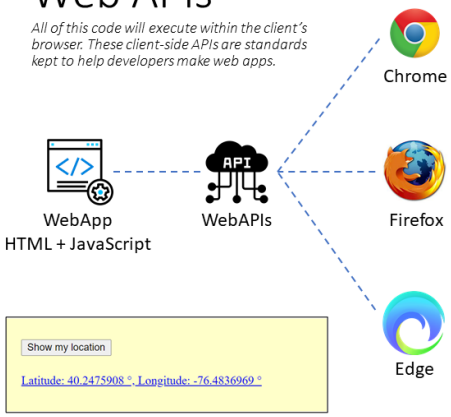
- Software packages often describe their functionality in terms of an API
  - How to execute or interact with the code
  - functions, objects, and methods available
  - "API reference" is synonymous with "user's manual"
- pandas API abstractions
  - 'data frames'
    - Easily manipulate data
    - Useful in statistics, machine learning, scientific computing, other data intense fields
  - Graphing
    - Ties into matplotlib
    - User need not worry about pandas / matplotlib communication
  - Prepped to anticipate changes
    - Deprecation warnings
    - Make new features immediately usable
  - Security
    - public vs private methods
    - Dev can keep consuming API while code versions are increasing
- No HTTP on this slide

Pandas API Reference:
https://pandas.pydata.org/docs/reference

Moraa Onwonga
moraa.onwonga@accenturefederal.com

Web APIs

All of this code will execute within the client's browser. These client-side APIs are standards kept to help developers make web apps.



Web app Utilizing Geolocation Web API to retrieve a user's location and display it on a page.

- Typically front-end development
  - Usually JavaScript app to browser (not always)
  - Extending functionality within browser; no HTTP transaction per se
- W3C tracks specs, guidelines, and web APIs
  - https://www.w3.org/TR/?tag=webapi
- Mozilla keeps a list of Web APIs
  - https://developer.mozilla.org/en-US/docs/Web/API

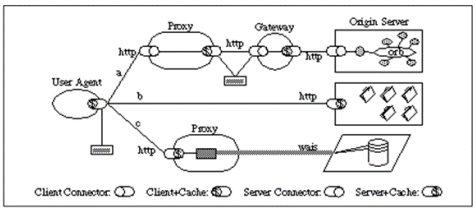
```
function geoFindMe() {

...

  function success(position) {
    const latitude  = position.coords.latitude;
    const longitude = position.coords.longitude;

    status.textContent = '';
    mapLink.href =
`https://www.openstreetmap.org/#map=18/${latitude}/${longitude}`;
    mapLink.textContent = `Latitude: ${latitude} °, Longitude:
${longitude} °`;
  }

  function error() {
    status.textContent = 'Unable to retrieve your location';
  }

  if(navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(success, error);
  }

}
```

Abbreviated JavaScript code utilizing the geolocation API. Dev can focus on writing apps, not on the specifics of common tasks.

Geolocation API:
https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API/Using_the_Geolocation_API

RESTful APIs

Web architecture style defined by a set of constraints applied to elements within the architecture, by Roy Thomas Fielding (2000).



A user agent is portrayed in the midst of three parallel interactions, (a, b, c). The interactions were not satisfied by the User Agent cache, so each request has been routed to a resource origin according to properties of each resource identifier, and the configuration of the client connector. Each component is only aware of the interaction with their own client or server connections.

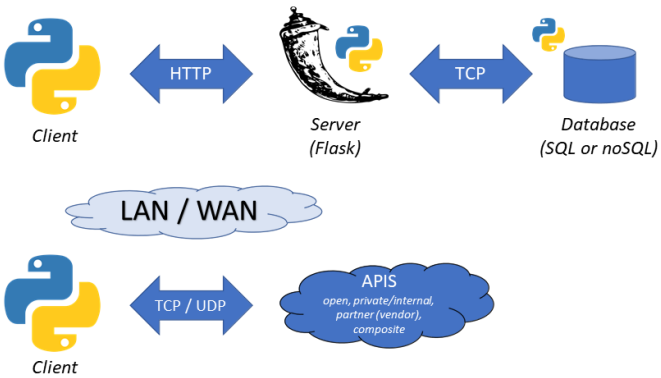
- Web *architecture* describing access to '*uniform connector interfaces*' (URIs)
  - Roy Thomas Fielding (2000)
  - https://roy.gbiv.com/pubs/dissertation/rest_arch_style.htm
- REST Architecture design constraints
  - Client-Server
  - Stateless communication
  - Responses may be cached
  - Uniform Interfaces (URIs) define server-side
  - Layered system (proxies)
  - Optional - Code on Demand
    - ok to return JavaScript to execute
- HTTP is *not* a design constraint

RESTful APIs:
https://roy.gbiv.com/pubs/dissertation/rest_arch_style.htm

APIs with Python

Python can be used to design the client, server, and / or the database. More generally, a Python client can be designed to interop with nearly any software API.

- Client
  - requests
  - Vendor libraries
- Server
  - Flask
  - Django
  - Bottle
  - Tornado
  - CherryPy
  - ... and more
- Database
  - SQLite3
  - Bindings to all DBs



Moraa Onwonga
moraa.onwonga@accenturefederal.com

10. Object Oriented Programming for APIs

Lab Objective

The most popular Python library for sending HTTP requests and processing the subsequent HTTP response is the *requests* library! Using simple functions, we are able to send requests to APIs (among other sources) and store the returned information in a single variable. Let's explore the idea that a single object can hold a great deal of information (if you can access it correctly)!

Procedure

1. The *requests* library is so popular that there is a loud call for it to come pre-packaged with Python installation! Let's confirm that it is installed.

```
student@bchd:~$ python3 -m pip install requests
```

2. The Star Wars API (SWAPI) is a fun place to start, with "all the Star Wars data you've ever wanted: Planets, Spaceships, Vehicles, People, Films and Species from all SEVEN Star Wars films"! Let's create a basic Python script to connect to this API.

```
student@bchd:~$ vim swapidemo1.py
```

```
#!/usr/bin/env python3
"""Star Wars API HTTP response parsing"""

# requests is used to send HTTP requests (get it?)
import requests

URL= "https://swapi.dev/api/people/1"

def main():
    """sending GET request, checking response"""

    # SWAPI response is stored in "resp" object
    resp= requests.get(URL)

    # what kind of python object is "resp"?
    print("This object class is:", type(resp), "\n")

    # what can we do with it?
    print("Methods/Attributes include:", dir(resp))

if __name__ == "__main__":
    main()
```

3. This script will:

    - send a GET request to the Star Wars APIs
    - capture the HTTP response back from the API in the variable `resp`
    - print what kind of object `resp` is
    - show all methods and attributes the `resp` object possesses!

```
student@bchd:~$ python3 swapidemo1.py

This object class is: <class 'requests.models.Response'>

Methods/Attributes include: ['__attrs__', '__bool__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__', '__eq__',
'__exit__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__nonzero__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_content', '_content_consumed', '_next', 'apparent_encoding', 'close', 'connection',
'content', 'cookies', 'elapsed', 'encoding', 'headers', 'history', 'is_permanent_redirect', 'is_redirect', 'iter_content', 'iter_lines', 'json',
'links', 'next', 'ok', 'raise_for_status', 'raw', 'reason', 'request', 'status_code', 'text', 'url']
```

4. The class output ('requests.models.Response') shows us this object belongs to the Response class in the requests library. There is a great deal of information inside this single object- let's use some of the methods to view some of it.

```
student@bchd:~$ vim swapidemo2.py
```

```
#!/usr/bin/env python3
"""Star Wars API HTTP response parsing"""

# pprint helps make things like dictionaries more human-readable
from pprint import pprint

# requests is used to send HTTP requests (get it?)
import requests

URL= "https://swapi.dev/api/people/1"

def main():
    """dissecting a requests.Response object"""

    # SWAPI response is stored in "resp" object
    resp= requests.get(URL)

    print("STATUS CODE and REASON:", resp.status_code, resp.reason)

    # dict() will force resp.headers into a dictionary format pprint can use
    pprint(dict(resp.headers))

if __name__ == "__main__":
    main()
```

5. Now execute your code and review the output.

```
student@bchd:~$ python3 swapidemo2.py

STATUS CODE and REASON: 200 OK
{'Allow': 'GET, HEAD, OPTIONS',
 'Connection': 'keep-alive',
 'Content-Type': 'application/json',
 'Date': 'Mon, 20 Dec 2021 17:07:17 GMT',
 'ETag': '"ee398610435c328f4d0a4e1b0d2f7bbc"',
 'Server': 'nginx/1.16.1',
 'Strict-Transport-Security': 'max-age=15768000',
 'Transfer-Encoding': 'chunked',
 'Vary': 'Accept, Cookie',
 'X-Frame-Options': 'SAMEORIGIN'}
```

200 OK: our request was successful! This is always a good status code to have!

Allow: what methods are permitted against this API path.

Connection: controls whether the network connection stays open after the current transaction finishes. `keep-alive` means the connection stays open and more requests may follow.

Content-Type: tells the client what media type is contained in the content (JSON, in our case).

Date: date and time at which the message was originated.

ETag: "entity tag" which identifies whether the data was updated since the last request.

Server: the server that generated the HTTP response.

Strict-Transport-Security: tells browsers states access should be HTTPS only. `max-age` is the number of seconds that the browser should remember this fact.

Transfer-Encoding: the form of encoding used to safely transfer the payload body to the user. `chunked` means the data is sent in a series of chunks.

Vary: dictates whether a cached response is satisfactory or if a fresh response should be requested.

X-Frame-Options: indicate whether or not a browser should be allowed to render a page in a `<frame>`, `<iframe>`, `<embed>` or `<object>`. `SAMORIGIN` means the page can only be displayed in a frame on the same origin as the page itself.

6. It's good to be able to read the metadata contained inside a header. But we're here to be geeks, not nerds-- let's get that Star Wars data! Let's demonstrate the difference between the `.content` and `.text` attributes and the `.json()` method!

```
student@bchd:~$ vim swapidemo3.py
```

```
#!/usr/bin/env python3
"""Star Wars API HTTP response parsing"""

# pprint helps make things like dictionaries more human-readable
from pprint import pprint

# requests is used to send HTTP requests (get it?)
import requests

URL= "https://swapi.dev/api/people/1"

def main():
    """getting at the JSON attached to this response"""

    # SWAPI response is stored in "resp" object
    resp= requests.get(URL)

    x= """
    The .content attribute returns the content (our Star Wars data)!...
    but in bytes. Bytes are a sequence of bits/bytes that represent data,
    but is only really meant to be read by machines.

    Note the superfluous apostrophes (') and the "b" character at the beginning of each line.
    """

    print(x)
    print(type(resp.content))
    pprint(resp.content)
    input()

    y= """
    The .text attribute will return the content as a string! Much more readable!
    However, this data is useless to us in most programs...
    we can't easily parse strings!
    """

    print(y)
    print(type(resp.text))
    pprint(resp.text)
    input()

    z= """
    The .json() method is wonderful. If the page is returning JSON, the .json() method
    will convert it into the Pythonic data equivalent! We can now use this data
    INFINITELY more effectively because it has been converted to a Python dictionary!
    """

    print(z)
    print(type(resp.json()))
    pprint(resp.json())

    # now we can do some cool stuff with the data we received!
    print("\n" + resp.json()["name"] + " is the protagonist of Star Wars! He appeared in the following films:")

    for film in resp.json()["films"]:
        print("  •", requests.get(film).json()["title"])

if __name__ == "__main__":
    main()
```

7. Run this script. Press `ENTER` to move through the output; observe how each tool works and how the data is returned differently!

```
student@bchd:~$ python3 swapidemo3.py
```

8. Well done! `requests` is a powerful library, and hopefully the reasoning behind its popularity is more apparent to you!

11. Practical Application of Lists

Lab Objective

In this lecture we'll review the structure and nature of Python lists, as well as explore their necessity in working with APIs.

A list in Python would be very similar to a list you'd take to a grocery store; a collection of items written out in a certain order from first to last.

```
student@bchd:~$ python3

>>> groceries= ["chips", "broccoli", "ice cream"]
```

When I get my groceries, I always get my ice cream last- nobody wants melty ice cream before you even get out of the store! Even when there's only three items in the cart... there are three elements in this list, right?

```
>>> len(groceries)

3
```

Yup, three elements (items) in that `groceries` list. Suppose I wanted to print out the value of `ice cream` from the `groceries` list; as it is the third element, this should work.

```
>>> print("The last item I buy at the store is " + groceries[3])

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

WELL, that didn't work! But the reasoning behind it is simple; lists are written in order and they always start counting at ZERO!

```
# element #      0         1           2
# groceries= ["chips", "broccoli", "ice cream"]
```

So let's try that again.

```
>>> print("The last item I buy at the store is " + groceries[2])

The last item I buy at the store is ice cream
```

However, there is another way to go about SLICING (returning a small part of) lists. One can count UP (+1) through the index of a list, but one can also count DOWN (-1). Consider the diagram below:

```
# element   ->  0         1           2
# groceries= ["chips", "broccoli", "ice cream"]
# <- element   -3        -2          -1
```

So if we "count to the left," this means the last element will ALWAYS be -1.

```
>>> print("The last item I buy at the store is " + groceries[-1])

The last item I buy at the store is ice cream
```

Same output!

The other important thing to know about lists is that they are MUTABLE-- which is just a fancy way of saying that lists can *change*. Let's add another element to our `groceries` list with the *.append()* method.

```
>>> groceries.append("gum")
```

Check what that did to our list:

```
>>> groceries

['chips', 'broccoli', 'ice cream', 'gum']
```

So what would that mean if we ran that last print function again?

```
>>> print("The last item I buy at the store is " + groceries[-1])

The last item I buy at the store is gum
```

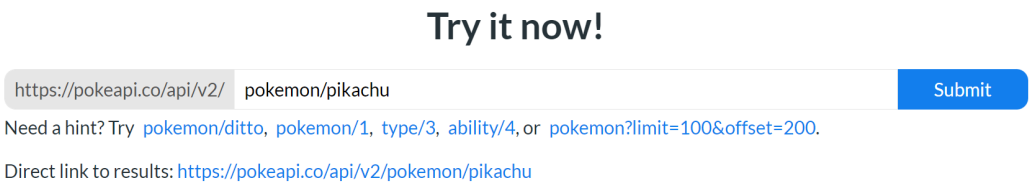
Lists' ordered index and mutability are important factors in how lists work. Python is an Objected Oriented Programming (OOP) language, and *everything* in Python is an object. Therefore, if a list is an ordered collection of objects, then pretty much anything can be part of a list!

>>> `grabbag= [123, 3.14, "alta3", ["sack lunch", "lantern", "elvish sword"], {"game title": "Zork", "date created": 1981}]

Can you count how many elements are inside of the list `grabbag`? ANSWER: 5! One integer, one float, one string, one list, and one dictionary!

So practically speaking, what does this have to do with APIs? APIs provide information, and much of the time that information is in JSON format. Visit the open API https://pokeapi.co/!

Change the text field to `pokemon/pikachu` and press the blue `Submit` button.



Scroll down to the section titled "Resource for pikachu". The *pokeapi* has an excellent feature in that it makes it easy to read the JSON being returned by this api. Find the link that reads `moves: [] 81 items`

Woah! Pikachu, a lightning mouse-monster that children worldwide hurl into battle every day, has 81 moves. If you were a fan looking to write an application that returned data about this character and each individual move it contained, you have quite a bit of data on your hands.

Resource for pikachu

This seems to be a list of dictionaries, each of which also contains NESTED dictionaries. If we are going to accurately access information we harvest from APIs, it certainly seems that lists will be a prominent part.

12. Lists

Lab Objective

The objective of this lab is to review lists, list methods, and built in functions. Lists are ways to organize data (items) within Python. An item may be just about any object you'd like (string, integer, float, other lists, dictionaries, and so on). Lists are ordered, which means if we place an item at index (position) '7' in the list, it should remain at index (position) '7'. Lists are also mutable, which means they can be altered (re-ordered, remove items, add items, etc.)

Strengthening our skills with Python lists will help us understand and work with JSON (and YAML).

Procedure

1. Let's stay in the habit of organizing our work. For now, make `~/mycode/` directory.

```
student@bchd:~$ mkdir ~/mycode/
```

2. Create a new script.

```
student@bchd:~$ vim ~/mycode/listrev01.py
```

3. Copy and paste the following into the script.

```
#!/usr/bin/python3
"""Learning or Reviewing about Lists | by Alta3 Research"""

def main():
    ## create an empty list
    myemptylist = []

    ## add to our list with a list method
    ## The extend method will add every item to the list
    myemptylist.extend('192.168.102.55')

    ## display our list
    print(myemptylist)

if __name__ == "__main__":
    main()
```

4. Save and exit with `:wq`

5. Run the script you just wrote.

```
student@bchd:~$ python3 ~/mycode/listrev01.py
```

6. Create a second new script.

```
student@bchd:~$ vim ~/mycode/listrev02.py
```

7. Copy and paste the following into the script.

```
#!/usr/bin/python3
"""Learning or Reviewing about Lists | by Alta3 Research"""

def main():
    anotheremptylist = []

    ## This will throw an ERROR
    ## the extend method expects exactly one argument
    anotheremptylist.extend('10.0.0.1', 'retro_game_server')

    print(anotheremptylist)

if __name__ == "__main__":
    main()
```

8. Save and exit with `:wq`

9. Run the script you just wrote to see this script **ERROR OUT**.

```
student@bchd:~$ python3 ~/mycode/listrev02.py
```

10. Create a third script.

```
student@bchd:~$ vim ~/mycode/listrev03.py
```

11. Copy and paste the following into the script.

```
#!/usr/bin/python3
"""Learning or Reviewing about Lists | by Alta3 Research"""

def main():
    ## create a list already containing IP addresses (strings)
    iplist = ['10.0.0.1', '10.0.1.1', '10.3.2.1']

    ## create a list of ports (strings)
    iplist2 = ['5060', '80', '22']

    ## display list
    print(iplist)

    ## Use the extend method on iplist, our list object
    ## Extend iterates over each 'thing' it is passed, and adds them to a list object
    iplist.extend(iplist2)

    ## show how iplist has changed
    print(iplist)

if __name__ == "__main__":
    main()
```

12. Save and exit with `:wq`

13. Run the script you just wrote.

```
student@bchd:~$ python3 ~/mycode/listrev03.py
```

14. Create a fourth script.

```
student@bchd:~$ vim ~/mycode/listrev04.py
```

15. Copy and paste the following into the script.

```
#!/usr/bin/python3
"""Learning or Reviewing about Lists | by Alta3 Research"""

def main():
    ## create a list already containing IP addresses (strings)
    iplist = ['10.0.0.1', '10.0.1.1', '10.3.2.1']

    ## create a list of ports (strings)
    iplist2 = ['5060', '80', '22']

    ## display list
    print(iplist)

    ## Use the append method on iplist, our list object
    ## append takes whatever it is passed and adds it to the list object (iplist)
    ## this will create a list within a list
    iplist.append(iplist2)

    ## show how iplist has changed
    print(iplist)

    ## just like extend, append expects exactly one item to be passed.
    ## If you'd like, uncomment the code below and see the error caused
    # iplist.append('aa:bb:cc:dd:ee:ff', '00:11:22:33:44:55')

if __name__ == "__main__":
    main()
```

16. Save and exit with `:wq`
17. Run the script you just wrote.

```
student@bchd:~$ python3 ~/mycode/listrev04.py
```

18. Review the script `listrev04.py`. If you'd like, uncomment the line at the end of the script with the remark that it will make the script fail, then run the script again. This fails because, just like `list.extend()`, it expects a single argument. `list.append()` also expects a single argument.
19. Create a fifth script.

```
student@bchd:~$ vim ~/mycode/listrev05.py
```

20. Copy and paste the following into the script.

```
#!/usr/bin/python3
"""Learning or Reviewing about Lists | by Alta3 Research"""

def main():
    ## a list of Alta3 classes
    alta3classes = ['python_basics', 'python_api_design', 'python_for_networking', 'kubernetes', \
      'sip', 'ims', '5g', '4g', 'avaya', 'ansible', 'python_and_ansible_for_network_automation']

    ## display the list
    print(alta3classes)

    ## how long is the list? use the built in len function
    ## THEN print (display) the results
    print(len(alta3classes))

    # display python_basics
    print(alta3classes[0])

    # display SIP
    print(alta3classes[4])

    # display Ansible
    print(alta3classes[9])

    ##Uncomment to see a list index out of range error
    #print(alta3classes[99])

    print(alta3classes[0:3])

    print(alta3classes[2:5])

    print(alta3classes[-1])

if __name__ == "__main__":
    main()
```

21. Save and exit with `:wq`
22. Run the script you just wrote.

```
student@bchd:~$ python3 ~/mycode/listrev05.py
```

23. Review the following rules about list slicing.
    - Basics of list slicing (this works for strings too). Pretend `a` is a list or string.

      ```
      a[start:end] # items start through end-1
      a[start:]    # items start through the rest of the array
      a[:end]      # items from the beginning through end-1
      a[:]         # a copy of the whole array
      ```

    - There is also the step value, which can be used with any of the above:

      ```
      a[start:end:step] # start through not past end, by step
      ```

    - The key point to remember is that the :end value represents the first value that is not in the selected slice. So, the difference between end and start is the number of elements selected (if step is 1, the default).
    - The other feature is that start or end may be a negative number, which means it counts from the end of the array instead of the beginning. So:

      ```
      a[-1]    # last item in the array
      a[-2:]   # last two items in the array
      a[:-2]   # everything except the last two items
      ```

    - Similarly, step may be a negative number:

      ```
      a[::-1]    # all items in the array, reversed
      a[1::-1]   # the first two items, reversed
      a[:-3:-1]  # the last two items, reversed
      a[-3::-1]  # everything except the last two items, reversed
      ```

24. After reviewing the rules, feel free to edit `~/mycode/listrev05.py` and play around with these rules regarding slicing.
25. Answer the following questions:
    - **Q: What are lists?**
        - *A: Lists offer users ways to create ordered data sets within Python.*
    - **Q: Do lists exist outside of Python?**
        - *A: Yes! In JSON and other programming languages they're called 'arrays'.*
26. Great job! That's it for this lab. If you're tracking your code in GitHub, issue the following commands:
    - `cd ~/mycode/`
    - `git add *`
    - `git commit -m "learning about lists"`
    - `git push origin`

13. Practical Application of Dictionaries

Lab Objective

In this lecture we'll review the structure and nature of Python dictionaries, as well as explore their necessity in working with APIs.

A dictionary is different than a list. Where the ORDER of elements in a list is an integral part of how lists work, dictionaries are NOT ordered. A dictionary like this:

```
student@bchd:~$ python3
>> dict1={"color":"red","ice cream":"chocolate","season":"fall"}
```

...is the same as this dictionary:

```
>> dict2={"season":"fall","ice cream":"chocolate","color":"red"}
```

This also means that when you are identifying values inside a dictionary, you do not use an index number. Instead, you pull values by their keys.

```
>> dict2["color"]
red
```

However, dictionaries are similar to lists in that they are MUTABLE (able to be changed). Using *assignment*, the values of dictionary keys can be changed:

```
>> dict2["color"]= "blue"
>> dict2["color"]
blue
```

the value of the key *color* has been changed to "blue."

Key/value pairs can be added to existing dictionaries through assignment or the `.update()` method.

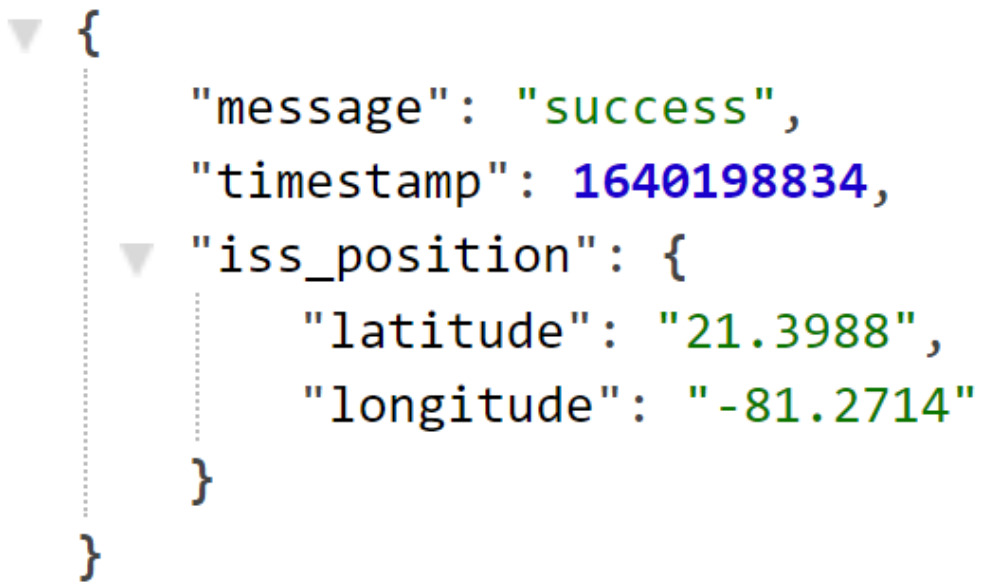
```
>> dict2["fruit"]= "apple"
>> dict2.update({"veggie":"broccoli"})
>> dict2
{'season': 'fall', 'ice cream': 'chocolate', 'color': 'red', 'fruit': 'apple', 'veggie': 'broccoli'}
```

Dictionary mutability also means key/value pairs can be removed as well.

```
>> dict2.pop("season")
>> del dict2["ice cream"]
>> dict2
{'color': 'red', 'fruit': 'apple', 'veggie': 'broccoli'}
```

Dictionaries are incredibly useful in Python just as they are in every other language. No other data structure is as accessible as dictionaries-- in lists, one must rely on indices which can be very error prone. Dictionaries are designed to find and pull data very quickly, which is why most data returned by APIs are in dictionary format.

Dictionaries can be absolutely massive, so let's look at one on the smaller side at http://api.open-notify.org/iss-now.json.



Here you can see that the data is highly organized and labeled. This API returns the current location of the International Space Station as it flies over earth. The `timestamp`, `latitude`, and `longitude` keys convey the data content very clearly, which is ideal when writing code that must access this information.

14. Dictionaries

Lab Objective

The objective of this lab is to refresh or introduce using python dictionaries, dictionary methods, and built in functions. Dictionaries are created with key:value pair relationships. Keys always recall values, but values will not recall keys. Dictionaries are unordered, which means if we place a key:value pair at index (position) '7' in the dictionary there is no guarantee it will remain at index (position) '7'. Dictionaries are also mutable, which means they can be altered (re-ordered, remove key:value pairs, add key:value pairs, etc.)

Strengthening our skills with Python dictionaries will help us understand and work with JSON (and YAML).

Procedure

1. Let's stay in the habit of organizing our work. For now, make `/home/student/mycode/` directory.

```
student@bchd:~$ mkdir ~/mycode/
```

2. Move to the `/home/student/mycode/` directory.

```
student@bchd:~$ cd ~/mycode/
```

3. Create a new script named `dictrev01.py`.

```
student@bchd:~$ vim ~/mycode/dictrev01.py
```

4. Copy and paste the following into the script.

```
#!/usr/bin/python3
"""Learning about Dictionaries | Alta3 Research"""

def main():
    """runtime code"""
    hostipdict = {'host01':'10.0.2.3', 'host02':'192.168.3.3', 'host03':'72.4.23.22'}

    ## Display the current state of our dictionary
    print(hostipdict)

    ## add another entry to the dict
    hostipdict['host04'] = '10.23.43.224'

    ## display the dict with the new entry for host4
    print(hostipdict)

    ## rewrite the value for host02
    hostipdict['host02'] = '192.168.70.55'

    ## display the dict with the new entry applied
    print(hostipdict)

    ## recall from the dict
    ## 'host02' should now point to '192.168.70.55'
    print(hostipdict['host02'])

    ## This will cause a key error
    ## toast01 is not a key
    ##print(hostipdict['toast01'])

if __name__ == "__main__":
    main()
```

5. Save and exit with `:wq`

6. Run the script you just wrote.

```
student@bchd:~$ python3 ~/mycode/dictrev01.py
```

7. Edit `~/mycode/dictrev01.py` and uncomment the last line of code. Run the script again and the script should fail with a key error. That is because the key `toast01` does not exist.

8. Create a second new script. In this one we'll try using a dictionary method called `.get()` to recall our data. This method expects to be given the key you are looking for. If it does not exist, it will **not** error out.

```
student@bchd:~$ vim ~/mycode/dictrev02.py
```

9. Copy and paste the following into the script.

```
#!/usr/bin/python3
"""Reviewing how to work with dictionaries | Alta3 Research"""

def main():
    firewalldict = {'sip':'5060', 'ssh':'22', 'http':'80'}

    ## display the current state of our dictionary
    print(firewalldict)

    ## add another entry to the dict
    ## notice that https maps to an INT, not a STRING
    firewalldict['https'] = 443

    ## display the dict with the new entry for host4
    print(firewalldict)

    ## display some dictionary data
    print('The print statement can be passed multiple items, provided they are separated by commas')
    print("The port in use for HTTP Secure is:", firewalldict['https'])

    ## this SHOULD fail but it will not because we are using the .get method
    print("A safer way to recall that data is to use the .get method:", \
      firewalldict.get('razzledazzlerootbeer'))

    ## use the .keys method to return a list of keys
    print(firewalldict.keys())

    ## use the .values method to return a list of values
    print(firewalldict.values())

    ## remove a single key from the dict
    del firewalldict["sip"]
    print(firewalldict)

if __name__ == "__main__":
    main()
```

10. Read through the comments, then save and exit.

11. Run the script you just wrote. It should work.

```
student@bchd:~$ python3 ~/mycode/dictrev02.py
```

12. Create a third script. In this script, we'll start to focus on methods a bit more.

```
student@bchd:~$ vim ~/mycode/dictrev03.py
```

13. To understand what a method is, you have to understand just a bit about classes and objects. Classes create objects. We won't get into creating classes here, but think of class like factory that can create objects. For example, there is a `dict` class that creates objects we call dictionaries. Functions can be defined inside of a class, which allows objects instantiated from that class to inherit those functions. We call these inherited functions, 'methods'.

14. Copy and paste the following into the script.

```
#!/usr/bin/python3
"""exploring dictionary methods | Alta3 Research"""

def main():
    """run time code"""
    vendordict = {'cisco': True, 'juniper': False, 'arista': True, 'netgear': True}
    custlist = ['acme', 'globex corporation', 'soylent green', 'initech', 'umbrella corporation']

    ## Display the current state of our dictionary
    print(vendordict)

    ## display all of the dictionary methods
    ## focus on the ones without underscores
    ## dict is a special word that Python treats as a dictionary
    ## FYI -- dict would be a terrible variable name
    print(dir(dict))
    # ['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', \
    # 'update', 'values']

    ## use a few dictionary methods
    print(vendordict.keys())
    print(vendordict.values())
    print(vendordict.get('juniper'))
    ## remove the key:value pair for netgear
    vendordict.pop('netgear')
    ## notice that 'netgear' no longer returns a value (the key:value pair is gone)
    print(vendordict.get('netgear'))

    ## display all of the list methods-- focus on the ones without underscores
    ## list is a special word that Python treats as a list
    ## FYI -- list would be a terrible variable name
    print(dir(list))
    # ['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', \
    # 'reverse', 'sort']
    custlist.append('cyberdyne')

    ## cyberdyne should now be part of the list
    print(custlist)

if __name__ == "__main__":
    main()
```

15. Read over the comments in the code, then save and exit with `:wq`
16. Run the script you just wrote. Compare the output to the code and figure out what each of the displayed lines represent.

```
student@bchd:~$ python3 ~/mycode/dictrev03.py
```

17. Great job! That's it for this lab. If you're tracking your code in GitHub, issue the following commands:
    - `cd ~/mycode/`
    - `git add *`
    - `git commit -m "learning about dictionaries"`
    - `git push origin`

15. List and Dict Modeling

Lab Objective

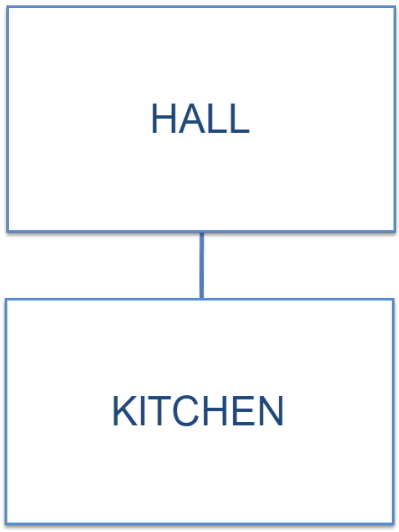
Learning how to work with functions and Python's data structures (lists and dictionaries) is unbelievably important. Why? Well, Network Devices are real world objects that need to be modeled before they can be automated.

A real world example might be to consider YANG (IETF RFC 6020). YANG is a data modeling structure that is used to model interfaces and configuration data of routers and switches. The protocol agnostic YANG model can then be transformed into data structures things like XML, JSON, or even Python data structures. Being able to look at and understand the relationship of data sets is a muscle we need to keep exercising.

In this lab we'll practice using Python data structures to model real-world objects. Our goal will be to practice the following basic skill sets: functions, dictionaries, lists, loops, and conditionals.

Procedure

1. Imagine two rooms in the real world. Could we model them with a Python data structure?



2. Consider the following dictionary, it describes the relationship of our two rooms:

```
## A dictionary linking a room to other rooms
rooms = {

            'Hall' : {
                  'south' : 'Kitchen'
                },

            'Kitchen' : {
                  'north' : 'Hall'
                }

          }
```

3. Make a new directory to work in.

```
student@bchd:~$ mkdir -p /home/student/mycode/
```

4. Move into `/home/student/mycode/`

```
student@bchd:~$ cd /home/student/mycode/
```

5. Create a script called, `mygame01.py`

```
student@bchd:~/mycode$ vim ~/mycode/mygame01.py
```

6. Create the following script:

```
#!/usr/bin/python3
"""Driving a simple game framework with
   a dictionary object | Alta3 Research"""

def showInstructions():
    """Show the game instructions when called"""
    #print a main menu and the commands
    print('''
    RPG Game
    ========
    Commands:
      go [direction]
      get [item]
    ''')

def showStatus():
    """determine the current status of the player"""
    # print the player's current location
    print('---------------------------')
    print('You are in the ' + currentRoom)
    # print what the player is carrying
    print('Inventory:', inventory)
    # check if there's an item in the room, if so print it
    if "item" in rooms[currentRoom]:
      print('You see a ' + rooms[currentRoom]['item'])
    print("---------------------------")



# an inventory, which is initially empty
inventory = []

# a dictionary linking a room to other rooms
rooms = {

            'Hall' : {
                  'south' : 'Kitchen'
                },

            'Kitchen' : {
                  'north' : 'Hall'
                }

         }

# start the player in the Hall
currentRoom = 'Hall'

showInstructions()

# breaking this while loop means the game is over
while True:
    showStatus()

    # the player MUST type something in
    # otherwise input will keep asking
    move = ''
    while move == '':
        move = input('>')

    # normalizing input:
    # .lower() makes it lower case, .split() turns it to a list
    # therefore, "get golden key" becomes ["get", "golden key"]
    move = move.lower().split(" ", 1)

    #if they type 'go' first
    if move[0] == 'go':
        #check that they are allowed wherever they want to go
```

```
            if move[1] in rooms[currentRoom]:
                #set the current room to the new room
                currentRoom = rooms[currentRoom][move[1]]
            # if they aren't allowed to go that way:
            else:
                print('You can\'t go that way!')

        #if they type 'get' first
        if move[0] == 'get' :
            # make two checks:
            # 1. if the current room contains an item
            # 2. if the item in the room matches the item the player wishes to get
            if "item" in rooms[currentRoom] and move[1] in rooms[currentRoom]['item']:
                #add the item to their inventory
                inventory.append(move[1])
                #display a helpful message
                print(move[1] + ' got!')
                #delete the item key:value pair from the room's dictionary
                del rooms[currentRoom]['item']
            # if there's no item in the room or the item doesn't match
            else:
                #tell them they can't get it
                print('Can\'t get ' + move[1] + '!')
```

7. So our goal isn't to write an RPG game, but there is the skeleton of one. Currently, our real world object is `rooms`. Review the code until you're certain you understand it. Ask the instructor for some help if anything is unclear.

8. Change permissions on your code so it is executable *Note: This chmod step is a Linux best practice step; feel free to do this to any of your scripts, if you wish.*

```
student@bchd:~/mycode$ chmod u+x ~/mycode/mygame01.py
```

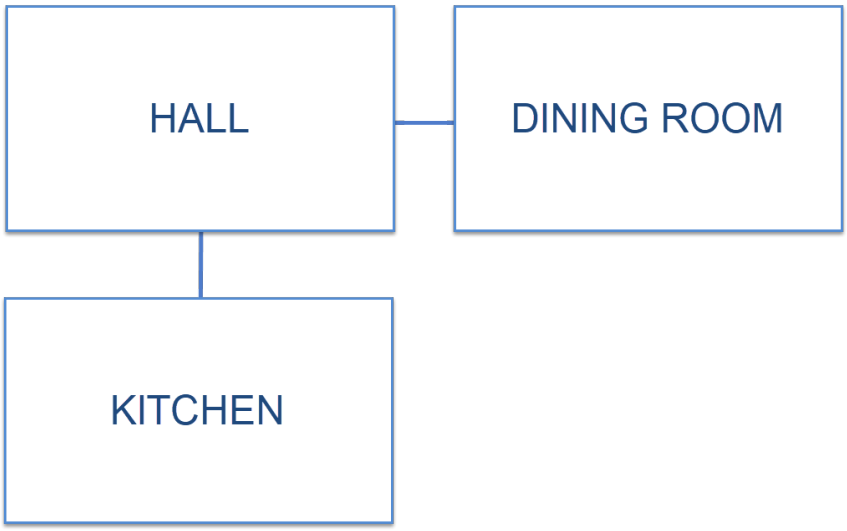
9. Run your code.

```
student@bchd:~/mycode$ python3 ~/mycode/mygame01.py
```

10. You should be able to `go south` and `go north`, but that's it. For 'fun', try to `go west`. You'll get a failure message.

11. As we discussed, if you find the `rooms` variable, you'll find that the map is a dictionary of rooms.

12. Let's alter the dictionary so that our map looks like the following:



13. You need to add a third room called the dining room. You also need to link it to the hall to the west. You also need to add data to the hall so that you can move to the dining room to the east. Edit the room dictionary so it looks like the following.

```
## A dictionary linking a room to other rooms
rooms = {

            'Hall' : {
                  'south' : 'Kitchen',
                  'east'  : 'Dining Room'
                },

            'Kitchen' : {
                  'north' : 'Hall'
                },
            'Dining Room' : {
                  'west' : 'Hall'
             }
          }
```

14. Try out the new game with your changes. See the instructor if you're having trouble when you `go east` from the hall.

15. Let's try to add an item laying in the hall, that item will be a `key`.

```
## A dictionary linking a room to other rooms
rooms = {

            'Hall' : {
                  'south' : 'Kitchen',
                  'east'  : 'Dining Room',
                  'item'  : 'key'
                },

            'Kitchen' : {
                  'north' : 'Hall'
                },
            'Dining Room' : {
                  'west' : 'Hall'
             }
          }
```

16. After this change you should be able to use the command `get key` when 'You see a key' (in the Hall).

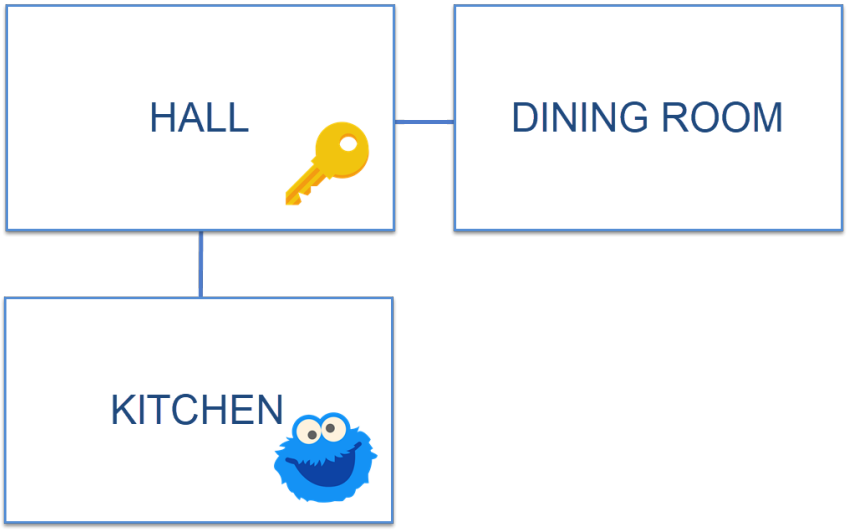
17. So far, working with data structures seems pretty easy. So, let's add a monster in the kitchen.

```
## A dictionary linking a room to other rooms
rooms = {

            'Hall' : {
                  'south' : 'Kitchen',
                  'east'  : 'Dining Room',
                  'item'  : 'key'
                },

            'Kitchen' : {
                  'north' : 'Hall',
                  'item'  : 'monster',
                },
            'Dining Room' : {
                  'west' : 'Hall'
             }
          }
```

18. Let's make the game end if a user enters a room with a monster. Add the following code to the very bottom of your script.

```
## If a player enters a room with a monster
if 'item' in rooms[currentRoom] and 'monster' in rooms[currentRoom]['item']:
    print('A monster has got you... GAME OVER!')
    break
```

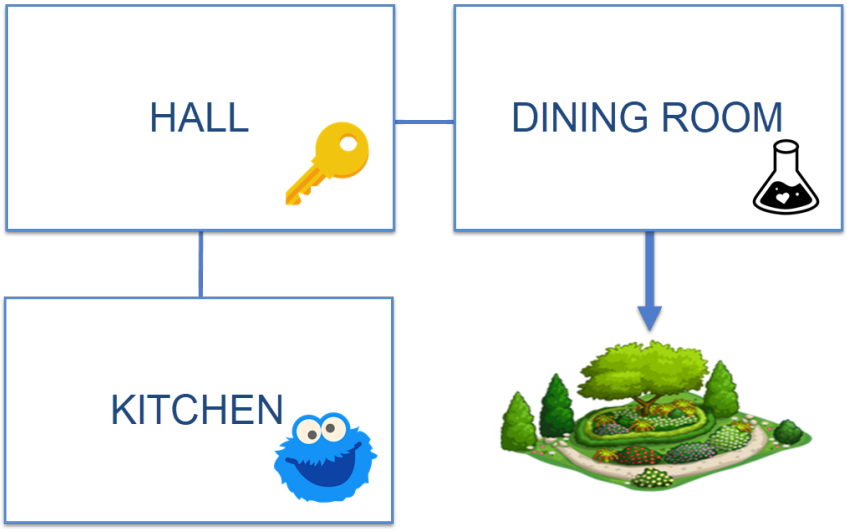
19. Confirm that the game works 'as designed' so far. For help, here's a map! Be careful, it's dangerous out there!



20. If your program is still working, let's wrap up our code. First add a garden south of the dining room- link the dining room to it and add a potion to the dining room.

```
## A dictionary linking a room to other rooms
rooms = {

            'Hall' : {
                  'south' : 'Kitchen',
                  'east'  : 'Dining Room',
                  'item'  : 'key'
                },

            'Kitchen' : {
                  'north' : 'Hall',
                  'item'  : 'monster',
                },
            'Dining Room' : {
                  'west' : 'Hall',
                  'south': 'Garden',
                  'item' : 'potion'
               },
            'Garden' : {
                  'north' : 'Dining Room'
            }
        }
```

21. Just to be clear, the following is what you have built:



22. Add the following code snippet to the **end of the game**. This will allow the user to win when they reach the garden with the potion and the key.

```
## Define how a player can win
if currentRoom == 'Garden' and 'key' in inventory and 'potion' in inventory:
    print('You escaped the house with the ultra rare key and magic potion... YOU WIN!')
    break
```

23. Edit the function `showInstructions()` to include more information on how to win the game, something like, "Get to the Garden with a key and a potion to win! Avoid the monsters! Commands include go *direction* and get *item*.

24. Save your code as `/home/student/mycode/mygame01.py`

25. Run your code and ensure it works. You might have to run a few times to ensure there are not any bugs.

26. **CODE CUSTOMIZATION 01 (OPTIONAL)** - Add *at least* one more room to the game (dictionary). This is much easier if you draw out what you're trying to do before you start doing.

27. **CODE CUSTOMIZATION 02 (OPTIONAL)** - Make your program more configurable by placing the data structure for `rooms` in an external file. Read that file into your program when the code is run.

28. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode/`
    - `git add *`
    - `git commit -m "Python and JSON practice, modeling with lists (arrays) and dicts (objects)"`
    - `git push origin`

16. Your First API Request

Lab Objective

The objective of this lab is to start to learn how to put together HTTP requests using the requests library. This is the most popular Python library for sending HTTP requests and processing the subsequent HTTP responses. In this lab we will make several GET requests from the Star Wars API (SWAPI).

Procedure

1. Start off in the home directory.

```
student@bchd:~$ cd
```

2. The *requests* library is so popular that there is a loud call for it to come pre-packaged with Python installation! Let's confirm that it is installed.

```
student@bchd:~$ python3 -m pip install requests
```

3. Answer the following questions:

    - **Q: How can we figure out how the Star Wars API (swapi.dev) works?**
        - *A: The documentation page can be found at https://swapi.dev/documentation. Each API will be different, but they all should have documentation someplace.*
    - **Q: What is the purpose of the requests library?**
        - *A: This abstracts the difficulty of sending HTTP GET messages. It is a synchronous library (blocking), but can be used within a loop or in a thread, if you had multiple API lookups to perform. Both of these techniques are beyond the scope up this lab, but will be covered in later exercises.*
    - **Q: What is pip?**
        - *A: A tool for installing Python packages from pypi.org (by default) or any other repository you wish to point to (such as GitLab or GitHub).*
    - **Q: Where can I read about requests?**
        - *A: The best place to get started is the requests homepage: https://docs.python-requests.org/en/latest/*

4. Let's create a basic Python script to connect to this API and grab some info about one of the most famous villains of all time- DARTH VADER. Move back into your mycode directory. By the way, this next script will FAIL.

```
student@bchd:~/mycode$ vim swapi1.py
```

5. Create the following:

```
#!/usr/bin/env python3
"""Alta3 Research
   Star Wars API HTTP response parsing"""

# pprint makes dictionaries a lot more human readable
from pprint import pprint

# requests is used to send HTTP requests (get it?)
import requests

# The following URL is constructed incorrectly. It should be api/people/4/
URL= "https://swapi.dev/api/people/four"

def main():
    """sending GET request, checking response"""

    # SWAPI response is stored in "resp" object
    resp= requests.get(URL)

    # convert the JSON content of the response into a python dictionary
    vader= resp.json()

    pprint(vader)

if __name__ == "__main__":
    main()
```

6. Save and exit with `:wq`

7. Try running the script. It won't QUITE work.

```
student@bchd:~/mycode$ python3 swapi1.py
```

8. Expect an error such as the following:

```
{'detail': 'Not found'}
```

9. Oops! We've made a very common error; our URL path is incorrect. Before we fix it, let's use the `.status_code` attribute from our `request.Response` object for some simple error handling.

```
student@bchd:~/mycode$ vim swapi2.py
```

10. Create the following:

```
#!/usr/bin/env python3
"""Alta3 Research
   Star Wars API HTTP response parsing"""

# pprint makes dictionaries a lot more human readable
from pprint import pprint

# requests is used to send HTTP requests (get it?)
import requests

URL = "https://swapi.dev/luke/force"      # Comment out this line
# URL= "https://swapi.dev/api/people/4/"     # Uncomment this line

def main():
    """sending GET request, checking response"""

    # SWAPI response is stored in "resp" object
    resp= requests.get(URL)

    # check to see if the status is anything other than what we want, a 200 OK
    if resp.status_code == 200:
        # convert the JSON content of the response into a python dictionary
        vader= resp.json()
        pprint(vader)

    else:
        print("That is not a valid URL.")

if __name__ == "__main__":
    main()
```

11. Now let's try it again, but *again* it won't work.

```
student@bchd:~/mycode$ python3 swapi2.py
```

12. Expect a failure, such as the following:

```
That is not a valid URL.
```

13. Answer the following questions:

    - **Q: What is wrong?**
        - *A: The api, https://example.vader/luke/force/ is not a valid URI. It is not returning a 200, so we are triggering the 'else' statement.*
    - **Q: What is resp.status_code?**
        - *A: This is an attribute associated with the object, "resp". This attribute tracks the response code returned to the HTTP GET we sent to the URI.*

14. A definite improvement. But now let's actually return the data. Update the code to use the correct URL

```
student@bchd:~/mycode$ vim swapi2.py
```

15. Look around line 10. Make your code look like the snippet below, by commenting out the first line that starts URL, and un-commenting the next.

```
# URL= "https://example.vader/luke/force"  # Comment out this line
URL= "https://swapi.dev/api/people/4/"     # Uncomment this line
```

16. Save with `:wq` and try running your script again.

```
student@bchd:~/mycode$ python3 swapi2.py
```

17. Expect output like the following:

```
{'birth_year': '41.9BBY',
 'created': '2014-12-10T15:18:20.704000Z',
 'edited': '2014-12-20T21:17:50.313000Z',
 'eye_color': 'yellow',
 'films': ['https://swapi.dev/api/films/1/',
           'https://swapi.dev/api/films/2/',
           'https://swapi.dev/api/films/3/',
           'https://swapi.dev/api/films/6/'],
 'gender': 'male',
 'hair_color': 'none',
 'height': '202',
 'homeworld': 'https://swapi.dev/api/planets/1/',
 'mass': '136',
 'name': 'Darth Vader',
 'skin_color': 'white',
 'species': [],
 'starships': ['https://swapi.dev/api/starships/13/'],
 'url': 'https://swapi.dev/api/people/4/',
 'vehicles': []}
```

18. **CHALLENGE 1 (OPTIONAL)** - Using the data returned above, can you create the following string? Values returned from the data above are in {curly braces}. You might try using an 'f-string' to complete this challenge.

```
{Darth Vader} was born in the year {41.9BBY}. His eyes are now {yellow} and his hair color is {none}.
```

19. **CHALLENGE 2 (OPTIONAL)** - Same as above, but add the following line! You will need to send two additional GET requests to get the title/name from `https://swapi.dev/api/films/1/` and `https://swapi.dev/api/starships/13/`!

```
He first appeared in the movie {A New Hope} and could be found flying around in his {TIE Advanced x1}.
```

20. **CHALLENGE SOLUTION 01** - One possible solution to **CHALLENGE 01** is as follows:

```
#!/usr/bin/env python3
"""Alta3 Research
   CHALLENGE SOLUTION 01
   {Darth Vader} was born in the year {41.9BBY}. His eyes are now {yellow} and his hair color is {none}."""

# pprint makes dictionaries a lot more human readable
from pprint import pprint

# requests is used to send HTTP requests (get it?)
import requests

#URL = "https://swapi.dev/luke/force"      # Comment out this line
URL= "https://swapi.dev/api/people/4/"     # Uncomment this line

def main():
    """sending GET request, checking response"""

    # SWAPI response is stored in "resp" object
    resp= requests.get(URL)

    # check to see if the status is anything other than what we want, a 200 OK
    if resp.status_code == 200:
        # convert the JSON content of the response into a python dictionary
        vader= resp.json()
        pprint(vader)

    else:
        print("That is not a valid URL.")



    # CHALLENGE 01 - SOLUTION
    # Create the following:
    # {Darth Vader} was born in the year {41.9BBY}. His eyes are now {yellow} and his hair color is {none}

    # this solution uses "f-strings" (string templates) to return the data
    print(f"{vader['name']} was born in the year {vader['birth_year']}. His eyes are now {vader['eye_color']} and his hair color is
{vader['hair_color']}")

if __name__ == "__main__":
    main()
```

21. **CHALLENGE SOLUTION 02** - One possible solution to **CHALLENGE 02** is as follows:

```
#!/usr/bin/env python3
"""Alta3 Research
   CHALLENGE SOLUTION 02
   He first appeared in the movie {A New Hope} and could be found flying around in his {TIE Advanced x1}."""

# pprint makes dictionaries a lot more human readable
from pprint import pprint

# requests is used to send HTTP requests (get it?)
import requests

#URL = "https://swapi.dev/luke/force"      # Comment out this line
URL= "https://swapi.dev/api/people/4/"     # Uncomment this line

def main():
    """sending GET request, checking response"""

    # SWAPI response is stored in "resp" object
    resp= requests.get(URL)

    # check to see if the status is anything other than what we want, a 200 OK
    if resp.status_code == 200:
        # convert the JSON content of the response into a python dictionary
        vader= resp.json()
        pprint(vader)

    else:
        print("That is not a valid URL.")


    # CHALLENGE 02 - SOLUTION
    # Create the following:
    # He first appeared in the movie {A New Hope} and could be found flying around in his {TIE Advanced x1}.

    # pick up the name of the first movie
    resp = requests.get(vader['films'][0])
    first_movie = resp.json() # skip checking for the 200

    # pick up the name of his ship
    resp = requests.get(vader['starships'][0])
    ship = resp.json()

    # this solution uses "f-strings" (string templates) to return the data
    print(f"He first appeared in the movie {first_movie['title']} and could be found flying around in his {ship['name']}.")



if __name__ == "__main__":
    main()
```

22. If you're tracking in an SCM, and you'd like to backup your code, run the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "First API request"`
    - `git push origin`

17. Lecture - Python Data sets vs JSON

Lab Objective

The objective of this lab is to introduce the differences between Python data sets, and JSON.

Procedures

Python data types vs JSON **© Alta3 Research**

Python Datatypes and JSON Rules

Python Data Types
- Strings – 'single quotes'
  - Will accept "double quotes"
- Numbers
- Literals – capitalized
  - True
  - False
  - None
- Dictionary – { }
- List - [ ]

JavaScript Object Notation
https://datatracker.ietf.org/doc/html/rfc8259
- Strings – "double quotes"
  - Illegal to use 'single quotes'
- Numbers
- Literals – lowercase *only*
  - true
  - false
  - null
- Objects – { }
- Arrays – [ ]

Python to JSON conversions:
https://docs.python.org/3/library/json.html#py-to-json-table

Python data types vs JSON **© Alta3 Research**

Python to JSON

Python

```
{'name':
   {'first': 'Bob',
    'last': 'Smith'
   },
 'languages': ['SIP', 'Python', 'Go'],
 'employed': True,
 'age': 32,
 'pets': None,
}
```

JSON

```
{"name":
   {"first": "Bob",
    "last": "Smith"
   },
 "languages": ["SIP", "Python", "Go"],
 "employed": true,
 "age": 32,
 "pets": null,
}
```

Moraa Onwonga
moraa.onwonga@accenturefederal.com

json.dumps()

"I want to use Python data to create JSON"

- Python data to be *marshalled* to JSON
- Dump Python data to a JSON string
- Useful for attaching Python data to an HTTP request
- Similar to `json.dump()`, although this does *not* target a file object

```
>>> # Dump Python data to JSON String
>>> import json
>>> x = ['foo', {'bar': ('baz', None, 1.0, 2)}]
>>> json.dumps(x)
"["foo", {"bar": ["baz", null, 1.0, 2]}]"
```

JSON dumps:
https://docs.python.org/3/library/json.html#json.dumps

json.dump()

"I want to use Python data create JSON within a target file"

- Python data to be *marshalled* to JSON and written into a file
- Similar to `json.dumps()`, although this targets a file object

```
#!/usr/bin/python3

import json

# python object(dictionary) to be dumped
webster ={'emp1': {'name': 'Lisa',
'designation': 'programmer', 'age': '34',
'salary': '54000'}, 'emp2': {'name': 'Elis',
'designation': 'Trainee', 'age': '24',
'salary': '40000'}}

# the json file where the output must be stored
with open('webster.json', 'w') as jsonfile:
    json.dump(webster, jsonfile)
```

JSON dump:
https://docs.python.org/3/library/json.html#json.dump

json.loads()

"I have a JSON string and want it to become Python data"

- JSON to be *unmarshalled* to Python data
- Useful if you have JSON stripped off of a HTTP response

Python string containing legal JSON

```
>>> # Load JSON String into the Python runtime
>>> import json
>>> z = '["foo", {"bar": ("a3r", null, 1.0, 2)}]'
>>> json.loads(z)
['foo', {'bar': ['a3r', None, 1.0, 2]}]
```

Python list

JSON loads:
https://docs.python.org/3/library/json.html#json.loads

Moraa Onwonga
moraa.onwonga@accenturefederal.com

json.load()

"I have a JSON file and want its contents to become Python data"

- JSON within a file to be *unmarshalled* to Python data
- Useful if you have JSON in a file that you want to work with

```
#!/usr/bin/python3
# Load JSON file into the Python runtime


import json

with open('data.json', 'r') as jsonfile:
    x = json.load(jsonfile)

# x is now python data
print(x)
```

JSON load:
https://docs.python.org/3/library/json.html#json.load

Moraa Onwonga
moraa.onwonga@accenturefederal.com

18. Python Data to JSON file

Lab Objective

The objective of this lab is to use Python to create a file on the local system as well as start learning about JSON. Not so surprisingly, **J**ava**S**cript **O**bject **N**otation was inspired by a subset of the JavaScript programming language. JSON has long since become language agnostic and exists as its own standard by the IETF in RFC 8259 (https://tools.ietf.org/html/rfc8259)

People chiefly liked JSON because it was *REALLY* easy for machines to create and parse, it had minimal size requirements as compared to XML, and was *pretty okay* for humans to create and parse. JSON's inability to be *REALLY* easy for humans is what led to the creation of YAML, but we can focus on that later. For now, we'll write a script that leverages Python's native support for JSON via its standard library and push some data out to the screen.

Review the following Python to JSON term translation table. When we use Python to output our data to JSON these are the conversions that will take place.

Python	JSON
dict	object
list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null
"double" OR 'single' quotes	"double" quotes ONLY

Procedure

1. Let's stay in the habit of organizing our work. For now, make `/home/student/mycode/` directory.

```
student@bchd:~$ mkdir ~/mycode/
```

2. Move into the new folder.

```
student@bchd:~$ cd ~/mycode/
```

3. Create a new script.

```
student@bchd:~/mycode$ vim ~/mycode/makejson01.py
```

4. Copy and paste the following into the script.

```
#!/usr/bin/python3
"""Reviewing how to parse json | Alta3 Research"""

# JSON is part of the Python Standard Library
import json

def main():
    """runtime code"""
    ## create a blob of data to work with
    hitchhikers = [{"name": "Zaphod Beeblebrox", "species": "Betelgeusian"},
      {"name": "Arthur Dent", "species": "Human"}]

    ## display our Python data (a list containing two dictionaries)
    print(hitchhikers)

    ## open a new file in write mode
    with open("galaxyguide.json", "w") as zfile:
        ## use the JSON library
        ## USAGE: json.dump(input data, file like object) ##
        json.dump(hitchhikers, zfile)

if __name__ == "__main__":
    main()
```

5. Save and exit with `:wq`
6. Run the script you just wrote.

```
student@bchd:~/mycode$ python3 ~/mycode/makejson01.py
```

7. Ensure your script produced a JSON file. It should be located in the file where we ran the script.

```
student@bchd:~/mycode$ cat ~/mycode/galaxyguide.json
```

8. Create a second new script.

```
student@bchd:~/mycode$ vim makejson02.py
```

9. In this script we explore `json.dumps()` which expects a single argument (a list or dictionary), performs the JSON transformation, and returns that as a JSON string. Copy and paste the following into the script.

```
#!/usr/bin/python3
"""The json.dumps() function creates a JSON string | Alta3 Research"""

# JSON is part of the Python Standard Library
import json

def main():
    """runtime code"""
    ## create a blob of data to work with
    hitchhikers = [{"name": "Zaphod Beeblebrox", "species": "Betelgeusian"},
      {"name": "Arthur Dent", "species": "Human"}]

    ## display our Python data (a list containing two dictionaries)
    print(hitchhikers)

    ## Create the JSON string
    jsonstring = json.dumps(hitchhikers)

    ## Display a single string of JSON
    print(jsonstring)

if __name__ == "__main__":
    main()
```

10. Save and exit with `:wq`
11. Run the script you just wrote. Python and JSON are very much alike, but take a moment to study some of the differences.

```
student@bchd:~/mycode$ python3 ~/mycode/makejson02.py
```

12. Cool! Let's create a block of JSON that represents a data center layout. Create a new script.

```
student@bchd:~/mycode$ vim ~/mycode/datacenter.json
```

13. Copy and paste the following into your script:

```
{
    "row1": ["svralpha", "svrbeta", "svrgamma", "svrdelta"],
    "row2": ["svr-avengers", "svr-justlge"],
    "row3": ["svr1", "svr2b", "svr3c", "svr4d"]
}
```

14. Save and exit with `:wq`
15. Create a new script that can parse out our new JSON file.

```
student@bchd:~/mycode$ vim ~/mycode/makejson03.py
```

16. Copy and paste the following into the new script.

```
#!/usr/bin/python3
"""opening a static file containing JSON data | Alta3 Research"""

# JSON is part of the Python Standard Library
import json

def main():
    """runtime code"""
    ## open the file
    with open("datacenter.json", "r") as datacenter:
        datacenterstring = datacenter.read()

    ## display our decoded string
    print(datacenterstring)
    print(type(datacenterstring))
    print("\nThe code above is string data. Python cannot easily work with this data.")
    input("Press Enter to continue\n")

    ## Create the JSON string
    datacenterdecoded = json.loads(datacenterstring)

    ## This is now a dictionary
    print(type(datacenterdecoded))

    ## display the servers in the datacenter
    print(datacenterdecoded)

    ## display the servers in row3
    print(datacenterdecoded["row3"])

    ## display the 2nd server in row2
    print(datacenterdecoded["row2"][1])

    ## write code to
    ## display the last server in row3

if __name__ == "__main__":
    main()
```

17. Save and exit with `:wq`

18. Run your script!

```
student@bchd:~/mycode/$ python3 ~/mycode/makejson03.py
```

19. The script should display your data. The dictionary allows us to shout out a name of a row within the datacenter and get back a list of servers as a list. Because lists have order we can determine how the servers are arranged within a row. Try editing the code to return the last server in the 3rd row (this is `svr4d`).

20. With Python, there is always more than one way to acheive the same result. This time, instead of using **.loads()**, let's give **.load()** a try.

```
student@bchd:~/mycode$ vim ~/mycode/makejson04.py
```

```
#!/usr/bin/python3
"""opening a static file containing JSON data | Alta3 Research"""

# JSON is part of the Python Standard Library
import json

def main():
    """runtime code"""
    ## open the file
    with open("datacenter.json", "r") as datacenter:
        datacenterdecoded = json.load(datacenter)

    ## This is now a dictionary
    print(type(datacenterdecoded))

    ## display the servers in the datacenter
    print(datacenterdecoded)

    ## display the servers in row3
    print(datacenterdecoded["row3"])

    ## display the 2nd server in row2
    print(datacenterdecoded["row2"][1])

if __name__ == "__main__":
    main()
```

21. Run your improved script now!

```
student@bchd:~/mycode/$ python3 ~/mycode/makejson04.py
```

22. If you're tracking your code in a SCM, issue the following commands:

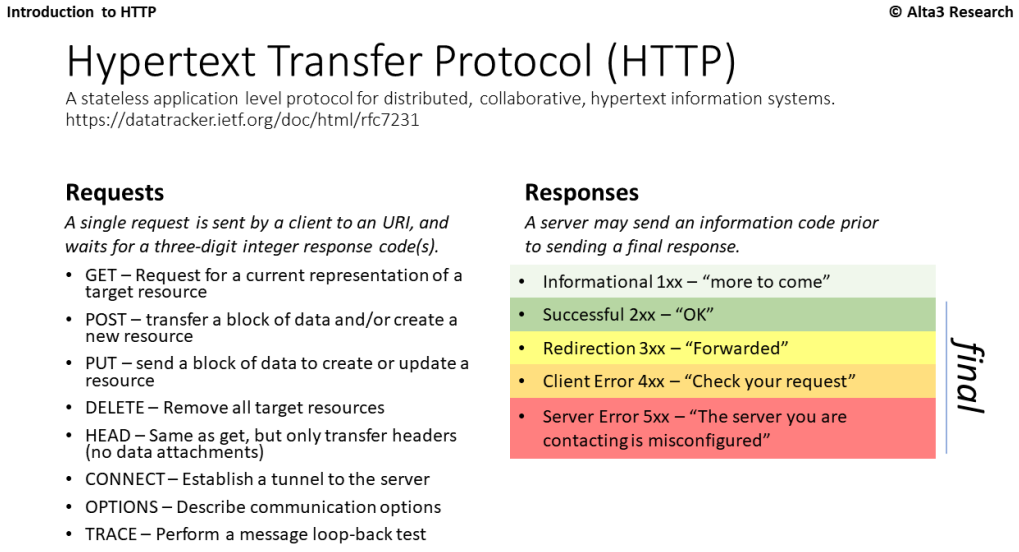
    - `cd ~/mycode/`
    - `git add *`
    - `git commit -m "reviewing how to manipulate JSON data with python"`
    - `git push origin HEAD`

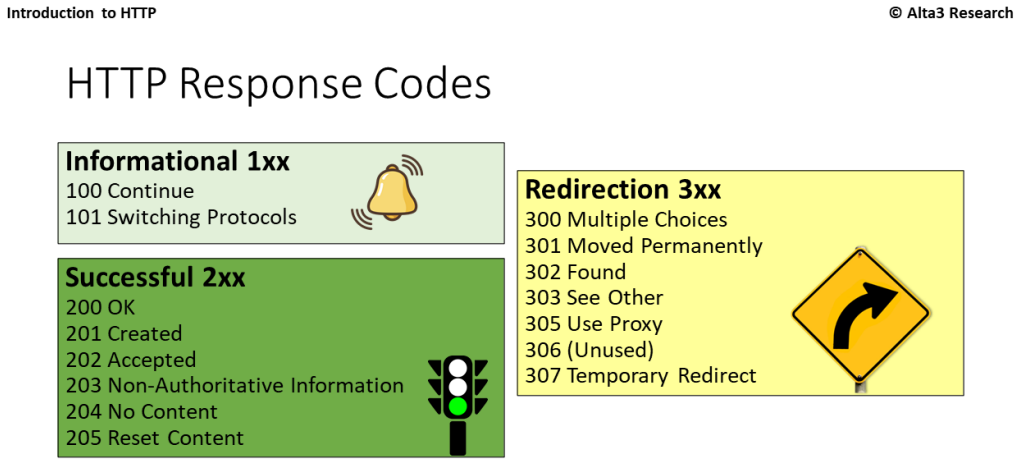
19. Lecture - Introduction to HTTP

Lab Objective

The objective of this lab is to introduce the HTTP protocol as a transport or access mechanism to RESTful APIs.

Procedures





Moraa Onwonga
moraa.onwonga@accenturefederal.com

HTTP Response Codes

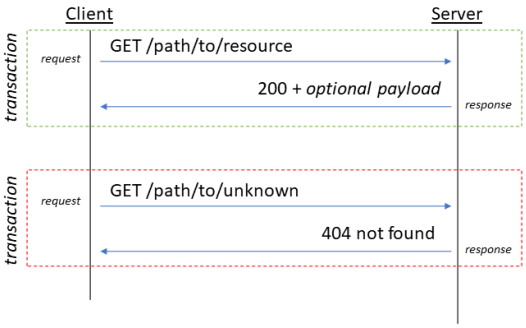
Client Error 4xx
400 Bad Request
402 Payment Required
403 Forbidden
404 Not Found
405 Method Not Allowed
406 Not Acceptable
408 Request Timeout
409 Conflict
410 Gone
411 Length Required
413 Payload Too Large

414 URI Too Long
415 Unsupported Media Type
417 Expectation Failed
426 Upgrade Required

Server Error 5xx
500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout
505 HTTP Version Not Supported

HTTP Transactions

An HTTP transaction is a request, any 1xx messages, and the single final response (2xx – 5xx) sent in response to the request.

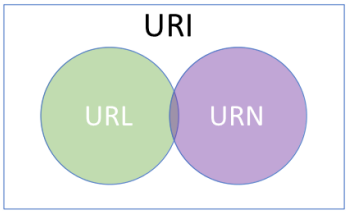


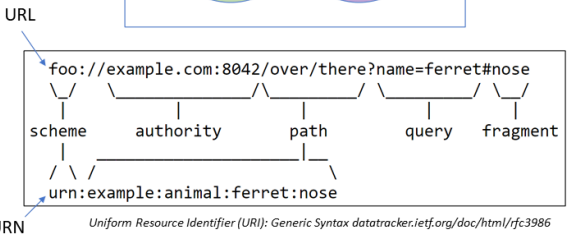
- An HTTP *client* or *user-agent* sends an HTTP request to an URI
  - Request may contain headers for authentication purposes
  - Some requests may have attachments, others may not
- Server sends a final response code
  - 200 may have optional payloads attached
    - HTML (webpages)
    - JSON
    - XML
    - TXT
  - Server sends "best fit"
  - Use codes for troubleshooting

HTTP Uniform Resource Identifiers (URIs)

Uniform Resource Identifier (URI): Generic Syntax
https://datatracker.ietf.org/doc/html/rfc3986

- Use the general term URI rather than the more restrictive terms URL or URN
- Uniform Resource Locator
  - Scheme – Identifies the protocol
  - Authority – Identifies the server (location)
  - Path – Identifies the resource
  - Query – key=value pairs appearing after a single ? separated by &
- Uniform Resource Name
  - Does not say "where" it is
  - Format: urn:NID:NSS
  - Unique names across space and time
    - urn:isbn:246013029
    - urn:mpeg:mpeg7:schema:2001
      - *Everything after the second colon is part of the NSS*
- IANA is the central authority that tracks names to ensure unique URLs and URNs





Uniform Resource Identifier (URI): Generic Syntax datatracker.ietf.org/doc/html/rfc3986

Moraa Onwonga
moraa.onwonga@accenturefederal.com

RESTful APIs serviced with HTTP

We will start with client-side requests to *open* APIs. The term *open* typically references publicly available resources with no, or limited authentication.

Most APIs return a dataset, such as JSON, on the 200 response. However, 200 responses will not always include attached data. API documentation will describe what you can expect to be, or not to be, returned.

Our Python client will:

1. Build HTTP GET request
   - Additional headers
   - ?query=parameters
2. Send HTTP request to target URI
3. Parse the response code
   - If a 200 strip off returned data (usually JSON) and *unmarshall*
   - Alert user to any non-2xx responses

Moraa Onwonga
moraa.onwonga@accenturefederal.com

20. Standard vs Third Party Libraries and Open APIs

Lab Objective

The objective of this lab is to compare using standard library Python tools (the `urllib.request` and `json` modules) and third party (the `requests` module). Though both are valid approaches to sending requests to APIs, you may find one preferable over the other.

Procedure

1. A web service has an address (url) just like a web page does. Instead of returning HTML for a web page it returns data. Open (http://api.open-notify.org/astros.json) in a web browser.

2. Take a moment to study our list of JSON-ized space heroes on the International Space Station. Way to go humanity!

3. You'll find JSON uses some new naming but you'll recognize the structure. It's dictionaries, key:value pairs, lists, strings, and integers. If you've been paying attention in Python class, JSON should be very clear. What is a bit tricky is that JSON LOOKS identical to Pythonic data structures, but it isn't. It's JSON. So, we'll need to convert it.

4. Let's stay in the habit of organizing our work.

```
student@bchd:~$ mkdir -p ~/mycode/iss/
```

5. Let's move into the `/home/student/mycode/iss` directory.

```
student@bchd:~$ cd ~/mycode/iss/
```

6. Open vim and add the following script:

```
student@bchd:~/mycode/iss$ vim ~/mycode/iss/ride_iss.py
```

```
#!/usr/bin/python3
"""Alta3 Research - astros on ISS"""

import urllib.request
import json

MAJORTOM = "http://api.open-notify.org/astros.json"

def main():
    """reading json from api"""
    # call the api
    groundctrl = urllib.request.urlopen(MAJORTOM)

    # strip off the attachment (JSON) and read it
    # the problem here, is that it will read out as a string
    helmet = groundctrl.read()

    # show that at this point, our data is str
    # we want to convert this to list / dict
    print(helmet)

    helmetson = json.loads(helmet.decode("utf-8"))

    # this should say bytes
    print(type(helmet))

    # this should say dict
    print(type(helmetson))

    print(helmetson["number"])

    # this returns a LIST of the people on this ISS
    print(helmetson["people"])

    # list the FIRST astro in the list
    print(helmetson["people"][0])

    # list the SECOND astro in the list
    print(helmetson["people"][1])

    # list the LAST astro in the list
    print(helmetson["people"][-1])

    # display every item in a list
    for astro in helmetson["people"]:
        # display what astro is
        print(astro)

    # display every item in a list
    for astro in helmetson["people"]:
        # display ONLY the name value associated with astro
        print(astro["name"])

if __name__ == "__main__":
    main()
```

7. Save and exit with `:wq`
8. Run your script.

```
student@bchd:~/mycode/iss$ python3 ~/mycode/iss/ride_iss.py
```

9. We've successfully accessed the API and returned the data as a Pythonic dictionary! But let's now accomplish the same goal using the `requests` library. If you haven't already, install the requests library.

```
student@bchd:~/mycode/iss$ python3 -m pip install requests
```

10. Write the following script:

```
student@bchd:~/mycode/iss$ vim ~/mycode/iss/requests-ride_iss.py

#!/usr/bin/python3
"""tracking the iss using
   api.open-notify.org/astros.json | Alta3 Research"""

# notice we no longer need to import urllib.request or json
import requests

## Define URL
MAJORTOM = 'http://api.open-notify.org/astros.json'

def main():
    """runtime code"""

    ## Call the webservice
    groundctrl = requests.get(MAJORTOM)
    # send a post with requests.post()
    # send a put with requests.put()
    # send a delete with requests.delete()
    # send a head with requests.head()

    ## strip the json off the 200 that was returned by our API
    ## translate the json into python lists and dictionaries
    helmetson = groundctrl.json()

    ## display our Pythonic data
    print("\n\nConverted Python data")
    print(helmetson)

    print('\n\nPeople in Space: ', helmetson['number'])
    people = helmetson['people']
    print(people)

if __name__ == "__main__":
    main()
```

11. Save and exit with `:wq`
12. Run your script.

```
student@bchd:~$ python3 ~/mycode/iss/requests-ride_iss.py
```

13. You should get the same output as if you used the `urllib.request` library, only this one took a lot fewer imports, and some less complicated lines of code!

CHALLENGE 01 - Tweak your script, and see if you can make it print out data in the following fashion.

```
People in space: 4
Eddie Kopra on the ISS
James Peake on the ISS
Yuri Kopra on the ISS
Buzz Aldrin on the ISS
```

Hint: Be sure to use the key 'name' and key 'craft' when you print() the above data.

SOLUTION 01- There are lots of ways you could solve this! Here is one way you could do it:

```
#!/usr/bin/python3
"""Alta3 Research - tracking ISS updated output"""

import urllib.request
import json

MAJORTOM = "http://api.open-notify.org/astros.json"

def main():
    """reading json from api"""
    # call the api
    groundctrl = urllib.request.urlopen(MAJORTOM)

    # strip off the attachment (JSON) and read it
    # the problem here, is that it will read out as a string
    helmet = groundctrl.read()

    helmetson = json.loads(helmet.decode("utf-8"))

    # display people in space
    print("People in space: " + str(helmetson["number"]))

    # display every item in a list
    for astro in helmetson["people"]:
        # display ONLY the name value associated with astro
        print(astro["name"] + " on the " + astro["craft"])

if __name__ == "__main__":
    main()
```

1. If you're tracking your code in an SCM, issue the following commands:
    - `cd ~/mycode/`
    - `git add *`
    - `git commit -m "Pulling in data from API with python standard library"`
    - `git push origin`

21. requests library - Open APIs

Lab Objective

The objective of this lab is to work with large JSON data sets returned by an API. At first, the data returned by an API may seem overwhelming. Techniques for conquering the "too much JSON data" may include: (1) limiting the amount of data returned by using query parameters, (2) using Python tools to comb through the data once it is returned, like the `dict.keys()` method, or by using loops.

The data set we will explore is extensive, and open (free). The data itself is sourced from the George R. R. Martin's Game of Thrones, available here

Within this lab we'll try out the following techniques:

- Reading the documentation to learn about the ways the service organizes and limits the data returned
- Using `dict.keys()` to only display relevant keys
- Looping across data returned
- Displaying with `pprint.pprint()` which is part of the standard library, and inserts newline characters to make data more human friendly

Resources:

- An API of Ice and Fire
- An API of Ice and Fire - Documentation
- Python Standard Library - Pretty Printer
- requests - Documentation

Procedure

1. To start, review the documentation on the APIs https://anapioficeandfire.com/Documentation
2. Create a directory to work in:

```
student@bchd:~$ mkdir ~/mycode/
```

3. Move to the `/home/student/mycode/` directory.

```
student@bchd:~$ cd ~/mycode/
```

4. Install the requests library. It is possible this package is already installed on your system, as it is a popular one.

```
student@bchd:~/mycode$ python3 -m pip install requests
```

5. Create a new script we can use to display the data returned by the root API.

```
student@bchd:~/mycode$ vim ~/mycode/iceAndFire01.py
```

6. Create the following script:

```
#!/usr/bin/python3
"""Alta3 Research - Exploring OpenAPIs with requests"""
# documentation for this API is at
# https://anapioficeandfire.com/Documentation

import requests

AOIF = "https://www.anapioficeandfire.com/api"

def main():
    ## Send HTTPS GET to the API of ICE and Fire
    gotresp = requests.get(AOIF)

    ## Decode the response
    got_dj = gotresp.json()

    ## print the response
    print(got_dj)

    ## display only the keys within
    ## the dictionary by using dict.keys()
    ## great for seeing what keys are available for lookup
    print(got_dj.keys())


if __name__ == "__main__":
    main()
```

7. Save and exit by pressing **Esc** and then `:wq`
8. Execute your script.

```
student@bchd:~/mycode$ python3 ~/mycode/iceAndFire01.py
```

9. Create a new script that we can use to explore the API relating to books.

```
student@bchd:~/mycode$ vim ~/mycode/iceAndFire02.py
```

10. Create the following script:

```
#!/usr/bin/python3
"""Alta3 Research - Exploring OpenAPIs with requests"""
# documentation for this API is at
# https://anapioficeandfire.com/Documentation

import pprint
import requests

AOIF_BOOKS = "https://www.anapioficeandfire.com/api/books"

def main():
    ## Send HTTPS GET to the API of ICE and Fire books resource
    gotresp = requests.get(AOIF_BOOKS)

    ## Decode the response
    got_dj = gotresp.json()

    ## print the response
    ## using pretty print so we can read it
    pprint.pprint(got_dj)

if __name__ == "__main__":
    main()
```

11. Save and exit by pressing **Esc** and then `:wq`
12. Execute your script.

```
student@bchd:~/mycode$ python3 ~/mycode/iceAndFire02.py
```

13. Create a new script that we can use to further explore the API relating to books. In this next script, we'll display the name, pages, ISBN, publisher, and number of characters found within the book.

```
student@bchd:~/mycode$ vim ~/mycode/iceAndFire03.py
```

14. Create the following script:

```
#!/usr/bin/python3
"""Alta3 Research - Exploring OpenAPIs with requests"""
# documentation for this API is at
# https://anapioficeandfire.com/Documentation

import requests

AOIF_BOOKS = "https://www.anapioficeandfire.com/api/books"

def main():
    ## Send HTTPS GET to the API of ICE and Fire books resource
    gotresp = requests.get(AOIF_BOOKS)

    ## Decode the response
    got_dj = gotresp.json()

    ## loop across response
    for singlebook in got_dj:
        ## display the names of each book
        ## all of the below statements do the same thing
        #print(singlebook["name"] + ",", "pages -", singlebook["numberOfPages"])
        #print("{}, pages - {}".format(singlebook["name"], singlebook["numberOfPages"]))
        print(f"{singlebook['name']}, pages - {singlebook['numberOfPages']}")
        print(f"\tAPI URL -> {singlebook['url']}\n")
        # print ISBN
        print(f"\tISBN -> {singlebook['isbn']}\n")
        print(f"\tPUBLISHER -> {singlebook['publisher']}\n")
        print(f"\tNo. of CHARACTERS -> {len(singlebook['characters'])}\n")

if __name__ == "__main__":
    main()
```

15. Save and exit with `:wq`

16. Execute your script.

```
student@bchd:~/mycode$ python3 ~/mycode/iceAndFire03.py
```

17. Now let's create a script that accepts input from the user and searches for a particular character. The API notes that we can send a parameter for each character's numeric entry.

```
student@bchd:~/mycode$ vim ~/mycode/iceAndFire04.py
```

18. Create the following script.

```
#!/usr/bin/python3
"""Alta3 Research - Exploring OpenAPIs with requests"""
# documentation for this API is at
# https://anapioficeandfire.com/Documentation

import requests
import pprint

AOIF_CHAR = "https://www.anapioficeandfire.com/api/characters/"

def main():
                ## Ask user for input
                got_charToLookup = input("Pick a number between 1 and 1000 to return info on a GoT character! " )

                ## Send HTTPS GET to the API of ICE and Fire character resource
                gotresp = requests.get(AOIF_CHAR + got_charToLookup)

                ## Decode the response
                got_dj = gotresp.json()
                pprint.pprint(got_dj)

if __name__ == "__main__":
                main()
```

19. Save and exit by pressing **Esc** and then `:wq`

20. Run your script.

```
student@bchd:~/mycode$ python3 ~/mycode/iceAndFire04.py
```

21. **CHALLENGE 01 (OPTIONAL) -** Return the house(s) affiliated with the character looked up, along with a list of books they appear in.

22. *SOLUTION 01 - One possible solution is as follows:*

```
#!/usr/bin/python3
"""Alta3 Research - RZFeeser
   SOLUTION 01 - Returning names to house and books with GOT API."""
# documentation for this API is at
# https://anapioficeandfire.com/Documentation

import requests
import pprint

AOIF_CHAR = "https://www.anapioficeandfire.com/api/characters/"

# check on the names of data passed
def name_finder(got_list):
    names = []  # list to return back of decoded names
    for x in got_list:
        # send HTTP GET to one of the entries within the list
        r = requests.get(x)
        decodedjson = r.json() # decode the JSON on the response
        names.append(decodedjson.get("name"))  # this returns the housename and adds it to our list
    return names  # when operation is over, send it back

def main():
        ## Ask user for input
        got_charToLookup = input("Pick a number between 1 and 1000 to return info on a GoT character! " )

        ## Send HTTPS GET to the API of ICE and Fire character resource
        gotresp = requests.get(AOIF_CHAR + got_charToLookup)

        ## Decode the response
        got_dj = gotresp.json()
        pprint.pprint(got_dj)

        # call our function
        print("This character belongs to the following houses:")
        for x in name_finder(got_dj.get("allegiances")):
            print(x)

        print("This character appears in the following books:")
        for x in name_finder(got_dj.get("books")):
            print(x)

if __name__ == "__main__":
    main()
```

23. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "Exploring how to work with large data sets"`
    - `git push origin`

22. requests library - RESTful GET and JSON parsing

Lab Objective

The objective of this lab is to learn to parse JSON attached to HTTP responses. This same skillet might be utilized in prepping JSON to attach to HTTP requests.

This is a critical skill that some students need additional help with. To that end, we'll use a highly available, ever-evolving, set of data from none other than the creators of Pokemon. No need to admit if you grew up watching the show, collecting cards, or playing the video games! Also, no need to know anything about Pokemon to appreciate this lab. At the end of the day, it is all about parsing JSON responses. It just so happens, https://pokeapi.co/ makes a very large open data-set available to us!

Procedure

1. Create a directory to work in.

```
student@bchd:~$ mkdir ~/mycode/pokemon/
```

2. Move to the `/home/student/mycode/` directory.

```
student@bchd:~$ cd ~/mycode/
```

3. Install the requests library. It is possible this package is already installed on your system, as it is a popular one.

```
student@bchd:~/mycode$ python3 -m pip install requests
```

4. Create a new script we can use to play around with the API found at https://pokeapi.co/ Check it out in a bowser before creating our script.

```
student@bchd:~/mycode$ vim pokemon/pikachu01.py
```

5. In this first script, we'll attempt to get all the names of the Pokemon available in the index. To do this, we'll augment the API with a `?limit=1000` parameter. We knew we could do this because we read the documentation. Create the following script:

```
#!/usr/bin/python3

import requests

# define base URL
POKEURL = "http://pokeapi.co/api/v2/pokemon/"

def main():

    # Make HTTP GET request using requests, and decode
    # JSON attachment as pythonic data structure
    # Augment the base URL with a limit parameter to 1000 results
    pokemon = requests.get(f"{POKEURL}?limit=1000")
    pokemon = pokemon.json()

    # Loop through data, and print pokemon names
    for poke in pokemon["results"]:
        # Display the value associated with 'name'
        #print(poke["name"])
        print(poke.get("name"))

    print(f"Total number of Pokemon returned: {len(pokemon['results'])}")

if __name__ == "__main__":
    main()
```

6. Save and exit with `:wq`

7. Execute your script.

```
student@bchd:~/mycode$ python3 pokemon/pikachu01.py
```

8. Great! Now let's try to work with the http://pokeapi.co/api/v2/items/ api. Create the following script. Let's write one that searches for the word 'heal' within the available items, and returns all words containing that word.

```
student@bchd:~/mycode$ vim pokemon/pikachu02.py
```

9. Create the following script.

```
#!/usr/bin/python3

import requests

ITEMURL = "http://pokeapi.co/api/v2/item/"

def main():

    # Make HTTP GET request using requests
    # and decode JSON attachment as pythonic data structure
    # Also, append the URL ITEMURL with a parameter to return 1000
    # items in one response
    items = requests.get(f"{ITEMURL}?limit=1000")
    items = items.json()

    # create a list to store items with the word "heal"
    healwords = []

    # Loop through data, and print pokemon names
    # item.get("results") will return the list
    # mapped to the key "results"
    for item in items.get("results"):
        # check to see if the current item's VALUE mapped to item["name"]
        # contains the word heal
        if 'heal' in item.get("name"):
            # if TRUE, add that item to the end of list healwords
            healwords.append(item.get("name"))

    ## list all
    print(f"There are {len(healwords)} words that contain the word 'heal' in the Pokemon Item API!")
    print("List of Pokemon items containing heal: ")
    print(healwords)

if __name__ == "__main__":
    main()
```

10. Save and exit with `:wq`

11. Execute your code.

```
student@bchd:~/mycode$ python3 pokemon/pikachu02.py
```

12. Let's make one last upgrade to our script. This time let's write a script that allows a user to pass in a word to search on, we count the number of times that word appears, print the total word list out to the screen, and then finally export everything to MS Excel XLSX format. To do this, we'll need to use `pandas`. Pandas also relies on `openpyxl` to export to XLSX format. Let's install both now.

```
student@bchd:~/mycode$ python3 -m pip install pandas openpyxl
```

13. Create a new script.

```
student@bchd:~/mycode$ vim pokemon/pikachu03.py
```

14. Write the following:

```
#!/usr/bin/python3

## for accepting arguments from the cmd line
import argparse

## for making HTTP requests
## python3 -m pip install requests
import requests

## for working with data in lots of formats
## python3 -m pip install pandas
import pandas

ITEMURL = "http://pokeapi.co/api/v2/item/"

def main():

    # Make HTTP GET request using requests
    # and decode JSON attachment as pythonic data structure
    # Also, append the URL ITEMURL with a parameter to return 1000
    # items in one response
    items = requests.get(f"{ITEMURL}?limit=1000")
    items = items.json()

    # create a list to store items with the word searched on
    matchedwords = []

    # Loop through data, and print pokemon names
    # item.get("results") will return the list
    # mapped to the key "results"
    for item in items.get("results"):
        # check to see if the current item's VALUE mapped to item["name"]
        # contains the search word
        if args.searchword in item.get("name"):
            # if TRUE, add that item to the end of list matchedwords
            matchedwords.append(item.get("name"))

    finishedlist = matchedwords.copy()
    ## map our matchedword list to a dict with a title
    matchedwords = {}
    matchedwords["matched"] = finishedlist

    ## list all words containing matched word
    print(f"There are {len(finishedlist)} words that contain the word '{args.searchword}' in the Pokemon Item API!")
    print(f"List of Pokemon items containing '{args.searchword}': ")
    print(matchedwords)

    ## export to excel with pandas
    # make a dataframe from our data
    itemsdf = pandas.DataFrame(matchedwords)
    # export to MS Excel XLSX format
    # run the following to export to XLSX
    # python -m pip install openpyxl
    # index=False prevents the index from our dataframe from
    # being written into the data
    itemsdf.to_excel("pokemonitems.xlsx", index=False)

    print("Gotta catch 'em all!")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Pass in a word to search\
    the Pokemon item API")
    parser.add_argument('--searchword', metavar='SEARCHW',\
    type=str, default='ball', help="Pass in any word. Default is 'ball'")
    args = parser.parse_args()
    main()
```

Save and exit with `:wq`

15.
16. Execute your code. Be sure to pass a search word at the command line! Search on the word 'heal', it should return 7 matching objects.

```
student@bchd:~/mycode$ python3 pokemon/pikachu03.py --searchword heal
```

17. Awesome! Way to go. Try it again with 'ball', it should return 33 matching objects.

```
student@bchd:~/mycode$ python3 pokemon/pikachu03.py --searchword ball
```

18. **CODE CUSTOMIZATION 01** - Use the PokeAPI to export a list of all Pokemon as plaintext, JSON, and Excel formats. *Hint: Explore the Pandas library for help on exporting datasets!*

19. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "learning to parse JSON responses within HTTP requests"`
    - `git push origin`

23. Lecture - APIs and JSON Decode

Lab Objective

The objective of this lab is to place more emphasis on the importance of the import statement in Python, as well as gain exposure to JSON and continue to learn how web APIs work.

This time we'll determine the current location of the international space station (ISS) via an API, and visualize that information for the user. The data sets used in this lab are very much accurate and in real-time.

Note: This is a demonstration that requires a GUI. As such, it will be performed by the instructor. If you would like to try this exercise, you need install Python and a Python IDE (such as PyCharm or Visual Studio) on your local machine.

Procedure

1. This ISS orbits the Earth every hour and a half. Wow. That's fast. To tell where it is at any given moment, visit this link: http://api.open-notify.org/iss-now.json
2. Create a place to work, such as **isslocation/**

1. Create the script, `iss_tracking.py`:

```
#!/usr/bin/python3

import urllib.request
import json

## Trace the ISS - earth-orbital space station
eoss = 'http://api.open-notify.org/iss-now.json'

## Call the webserv
trackiss = urllib.request.urlopen(eoss)

## put into file object
ztrack = trackiss.read()

## JSON 2 Python data structure
result = json.loads(ztrack.decode('utf-8'))

## display our Pythonic data
print("\n\nConverted Python data")
print(result)
input('\nISS data retrieved & converted. Press any key to continue')

location = result['iss_position']
lat = location['latitude']
lon = location['longitude']
print('\nLatitude: ', lat)
print('Longitude: ', lon)
```

2. Run the code. It should display the current location of the ISS.
3. It would be more useful if we were able to show the location of the space station on a map. First we'll need a map. We can get one from https://static.alta3.com/images/python/iss_map.gif

1. The map should be in the same directory as the script `iss_tracking.py`
2. The map is centered at 0,0 which is what we need.
3. We can place a `turtle` on our map to represent the ISS. What's a turtle? Check out this link to learn more about the turtle graphic module (which ships with Python3 install):
    - https://docs.python.org/3.3/library/turtle.html?highlight=turtle#module-turtle
4. Add the following line to the top of your code.

```
import turtle
```

5. Next we need to set the screen size to match the image, which is 720 x 360. Add the following code to the bottom of your code:

```
screen = turtle.Screen() # create a screen object
screen.setup(720, 360) # set the resolution
```

6. You want to be able to send the turtle to a particular latitude and longitude. To make this easy, set the screen to match the coordinates we are using. Add the following code to the bottom of your program.

```
screen.setworldcoordinates(-180,-90,180,90)
```

7. Add our map to the screen.

```
screen.bgpic('iss_map.gif')
```

8. Now to create an ISS sprite for turtle to use to depict the ISS. First, we'll download a tiny ISS icon to use. We can download one from https://static.alta3.com/images/python/spriteiss.gif

1. Add the following lines to the bottom of your code to register our tiny ISS, and place it on the screen.

```
screen.register_shape('spriteiss.gif')
iss = turtle.Turtle()
iss.shape('spriteiss.gif')
iss.setheading(90)
```

2. The ISS starts off in the center of the map, but now let's move it to the correct location on the map. Note, typically you report "Latitude, Longitude", but these are actually (x,y) screen coordinates. So they need to be 'reversed'.

```
lon = round(float(lon))
lat = round(float(lat))
iss.penup()
iss.goto(lon, lat)
turtle.mainloop()
```

3. At this time, your code should look like the following.

```
#!/usr/bin/python3

# standard library imports
import turtle
import urllib.request
import json

## Trace the ISS - earth-orbital space station
eoss = 'http://api.open-notify.org/iss-now.json'

## Call the webserv
trackiss = urllib.request.urlopen(eoss)

## put into file object
ztrack = trackiss.read()

## JSON 2 Python data structure
result = json.loads(ztrack.decode('utf-8'))

## display our Pythonic data
print("\n\nConverted Python data")
print(result)
input('\nISS data retrieved & converted. Press the ENTER key to continue')

location = result['iss_position']
lat = location['latitude']
lon = location['longitude']
print('\nLatitude: ', lat)
print('Longitude: ', lon)

screen = turtle.Screen() # create a screen object
screen.setup(720, 360) # set the resolution

screen.setworldcoordinates(-180,-90,180,90)

screen.bgpic('iss_map.gif')

screen.register_shape('spriteiss.gif')
iss = turtle.Turtle()
iss.shape('spriteiss.gif')
iss.setheading(90)

lon = round(float(lon))
lat = round(float(lat))

iss.penup()
iss.goto(lon, lat)
turtle.mainloop() # <-- this line should ALWAYS
  # be at the bottom of your script. It prevents the graphic from closing!!!
```

4. Test your program by running it. Wait a few seconds, then run it again to see where the ISS has moved to.

5. Pretty cool. Plot a point on our map to represent our city. Replace your code so it looks like the following. Notice that the code has been cleaned up a bit, but otherwise doing the same thing:

```
#!/usr/bin/python3

# standard library imports
import turtle
import urllib.request
import json

## Trace the ISS - earth-orbital space station
EOSS = 'http://api.open-notify.org/iss-now.json'

def main():
    ## Call the webserv
    trackiss = urllib.request.urlopen(EOSS)

    ## put into file object
    ztrack = trackiss.read()

    ## JSON 2 Python data structure
    result = json.loads(ztrack.decode('utf-8'))

    ## display our Pythonic data
    print("\n\nConverted Python data")
    print(result)
    input('\nISS data retrieved & converted. Press any key to continue')

    location = result['iss_position']
    lat = location['latitude']
    lon = location['longitude']
    print('\nLatitude: ', lat)
    print('Longitude: ', lon)

    screen = turtle.Screen()  # create a screen object
    screen.setup(720, 360)  # set the resolution

    screen.setworldcoordinates(-180,-90,180,90)

    screen.bgpic('iss_map.gif')

    ## My location
    yellowlat = 47.6
    yellowlon = -122.3
    mylocation = turtle.Turtle()
    mylocation.penup()
    mylocation.color('yellow')
    mylocation.goto(yellowlon, yellowlat)
    mylocation.dot(5)
    mylocation.hideturtle()

    ## ISS Sprite
    screen.register_shape('spriteiss.gif')
    iss = turtle.Turtle()
    iss.shape('spriteiss.gif')
    iss.setheading(90)

    lon = round(float(lon))
    lat = round(float(lat))
    iss.penup()
    iss.goto(lon, lat)
    turtle.mainloop()

if __name__ == "__main__":
    main()
```

6. Great. Let's create a final version that uses the requests library. In addition, we've added some functions to increase the overall re-usability of the code. Otherwise the code is still doing the same thing:

```
"""Alta3 Research | RZFeeser
   Visualizing tracking the ISS with open API data"""

#!/usr/bin/python3

# python3 -m pip install requests
import requests

# standard library imports
import turtle

## Trace the ISS - earth-orbital space station
EOSS = 'http://api.open-notify.org/iss-now.json'



# define a location on the map with a dot
def mapdot(yellowlat, yellowlon):
    mylocation = turtle.Turtle()
    mylocation.penup()
    mylocation.color('yellow')
    mylocation.goto(yellowlon, yellowlat)
    mylocation.dot(5)
    mylocation.hideturtle()
    return mylocation

# determine where the ISS is
def location():
    ## Call the webserv
    trackiss = requests.get(EOSS)

    # was a legal response code returned?
    if trackiss.status_code == 200:
        ## put into file object
        result = trackiss.json()
        ## determine latitude and longitude
        location = result.get('iss_position')  # preference for using the dict.get() method over key recall with []
        lat = location.get('latitude')
        lon = location.get('longitude')
        return (lat, lon)
    else:
        return None  # return None if the location cannot be determined



def main():
    loc = location()

    # stop execution of main if we cannot track the ISS
    if not loc:
        print("Unable to track ISS")
        return

    ## display our Pythonic data
    print("\n\nConverted Python data")
    print(loc)
    input('\nISS data retrieved & converted. Press any key to continue')
    lat,lon = loc
    print('\nLatitude: ', lat)
    print('Longitude: ', lon)

    ## prep the screen
    screen = turtle.Screen()  # create a screen object
    screen.setup(720, 360)  # set the resolution
    screen.setworldcoordinates(-180, -90, 180, 90) # describe the resolution as 2x as long as it is tall
    screen.bgpic('iss_map.gif')  # set the background image

    ## place a yellow dot at the location of the city we are currently in
    mapdot(47.6, -122.3)  # place got at this lat and lon
```

```
        ## Position the ISS Sprite
        screen.register_shape('spriteiss.gif')
        iss = turtle.Turtle()
        iss.shape('spriteiss.gif')
        iss.setheading(90)

        lon = round(float(lon))
        lat = round(float(lat))
        iss.penup()
        iss.goto(lon, lat)
        turtle.mainloop()

# call the main function
if __name__ == "__main__":
    main()
```

7. If you're tracking your code within an SCM, issue the following commands:

    - `cd ~/mycode/`
    - `git add *`
    - `git commit -m "ISS location lecture"`
    - `git push origin`

24. CHALLENGE - Key-pairs and HTTP GET

Lab Objective

The objective of this lab is to pose a challenge to students; design a Python client to communicate with an open APIs. The specific problem posed by this challenge is to determine the times of day when the ISS will be overhead, and potentially observable.

Procedure

1. Start in your home directory.

```
student@bchd:~$ cd
```

2. Create a place to work.

```
student@bchd:~$ mkdir ~/mycode/issloc/
```

3. Move to the `/home/student/mycode/issloc` directory.

```
student@bchd:~$ cd ~/mycode/issloc/
```

4. Now time to figure out the next time the ISS will be overhead. Copy and paste the following into a new browser tab:

    - http://api.open-notify.org/iss-pass.json *this will result in an error message*.

5. This API web service requires that we pass inputs. Inputs are passed via the URL we use to access the API resource. It's actually rather easy.

    - Inputs are added after a `?`
    - Inputs are separated with a `&`

6. The URL for Seattle, WA would be http://api.open-notify.org/iss-pass.json?lat=47.6&lon=-122.3. Edit the `lat` and `lon` values if you wish to reveal when the ISS will be overhead some other place. The JSON results back will look something like the following:

```
*** EXAMPLE ***
{
  "message": "success",
  "request": {
    "altitude": 100,
    "datetime": 1527395287,
    "latitude": 47.6,
    "longitude": -122.3,
    "passes": 5
  },
  "response": [
    {
      "duration": 641,
      "risetime": 1527406615
    },
    {
      "duration": 602,
      "risetime": 1527412410
    },
    {
      "duration": 261,
      "risetime": 1527418311
    },
    {
      "duration": 570,
      "risetime": 1527472536
    }
  ]
}
```

7. What you're looking at are multiple pass-over times. These are given in standard time format. If you haven't yet studied `time`, no big deal. You're looking at a 'standard time format.' It's also called Epoch time, Unix time, POSIX time, or UNIX Epoch time. It's all the same thing and is just a system for describing a point in time, which is defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970. It's easy to convert to human time... if you're a computer. Try some manual conversions of your `risetime` over here: https://www.epochconverter.com/

8. Create a Python script.

```
student@bchd:~/mycode/issloc$ vim /home/student/mycode/issloc/iss_overhead.py
```

9. The following **is not a solution**, but it is a template that will help get you started.

```
#!/usr/bin/python3
"""Alta3 Research | <your name here>
   Using an HTTP GET to determine when the ISS will pass over head"""

# python3 -m pip install requests
import requests

def main():
    """your code goes below here"""

    # stuck? you can always write comments
    # Try describe the steps you would take manually




if __name__ == "__main__":
    main()
```

10. **CHALLENGE 01** - Finish the python script, `iss_overhead.py` so that it alerts the user as to when the ISS will be over head.

11. **CHALLENGE 02** - Notice that the ovehead times are in Unix timestamps. Use the `time` library to decode the Unix timestamp to a current "human" time (*HINT: See* `time.ctime()`)

12. **CHALLNEGE 03** - Before the program begins, prompt the user for their current latitude and longitude. Remember a user might pass illegal coordinates, so you might want to validate the input before starting the lookup process.

1. If you're tracking your code within an SCM, issue the following commands:
    - `cd ~/mycode/`
    - `git add *`
    - `git commit -m "ISS challenge`
    - `git push origin`

25. Lecture - HTTP GET vs HTTP POST

Lab Objective

The objective of this lab is to illustrate the difference between an HTTP GET and HTTP POST.

Procedures

HTTP GET vs HTTP POST **© Alta3 Research**

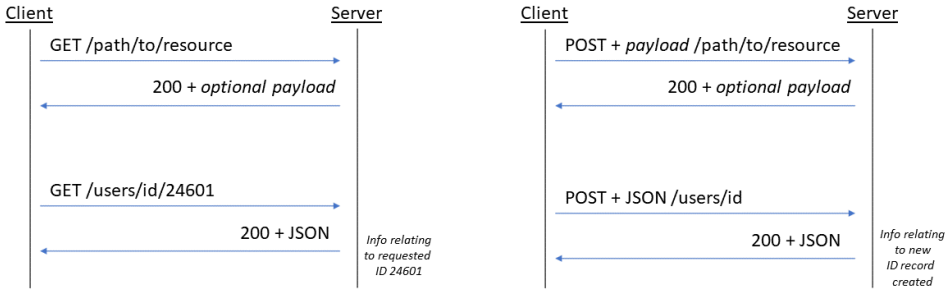
HTTP Requests

https://datatracker.ietf.org/doc/html/rfc7231#section-8.1.3

Method	Safe	Idempotent	Reference
CONNECT	no	no	Section 4.3.6
DELETE	no	yes	Section 4.3.5
GET	yes	yes	Section 4.3.1
HEAD	yes	yes	Section 4.3.2
OPTIONS	yes	yes	Section 4.3.7
POST	no	no	Section 4.3.3
PUT	no	yes	Section 4.3.4
TRACE	yes	yes	Section 4.3.8

HTTP GET vs HTTP POST **© Alta3 Research**

HTTP GET vs HTTP POST



GET is the primary mechanism of information retrieval. Sending a payload within a GET request message has no defined semantics; sending a payload body on a GET request may cause the request to be rejected.

POST a block of data (payload) such as fields of HTML form, or JSON. This should result in a non-idempotent behavior, such as creating a new record.

26. requests library - GET vs POST to REST APIs

Lab Objective

The objective of this lab is to learn the difference between a GET and POST request, and how they might be expected to interact differently with various RESTful APIs.

The default HTTP behavior of browsers (and most HTTP applications) is to send a GET. GETs require any information being 'passed' to the API to be done so within the URL. This is done via key=value pairs found after a `?` and delimited by `&` for example:
`www.example.com/search?subject=apis&date=recent`

This lab depends on the open APIs found at https://jsontest.com The site makes several APIs available, all of which return JSON. One API will be of particular interest to us, https://validate.jsontest.com which allows for the validation of JSON via a GET **or** POST. For now, just read over the documentation.

Procedure

1. Create a directory to work in.

   ```
   student@bchd:~$ mkdir ~/mycode/jsontest/
   ```

2. Move to the `/home/student/mycode/` directory.

   ```
   student@bchd:~$ cd ~/mycode/
   ```

3. Install the requests library. It is possible this package is already installed on your system, as it is a popular one.

   ```
   student@bchd:~/mycode$ python3 -m pip install requests
   ```

4. Create a new script we can use to play around with the IP API found at https://ip.jsontest.com This API returns JSON attached to an HTTP 200 response code revealing the sender's IP address.

   ```
   student@bchd:~/mycode$ vim jsontest/jsontestIP.py
   ```

5. Create the following script:

   ```
   #!/usr/bin/python3

   import requests

   # define the URL we want to use
   IPURL = "http://ip.jsontest.com/"

   def main():
       # use requests library to send an HTTP GET
       resp = requests.get(IPURL)

       # strip off JSON response
       # and convert to PYTHONIC LIST / DICT
       respjson = resp.json()

       # display our PYTHONIC data (LIST / DICT)
       print(respjson)

       # JUST display the value of "ip"
       print(f"The current WAN IP is --> {respjson['ip']}")

   if __name__ == "__main__":
       main()
   ```

6. Save and exit with `:wq`

7. Execute your script.

   ```
   student@bchd:~/mycode$ python3 jsontest/jsontestIP.py
   ```

8. The IP address should be displayed. Cool! Let's now try to validate some JSON.

```
student@bchd:~/mycode$ vim jsontest/jsontestValidateGET.py
```

This API works one of two ways. The first (which we'll explore now) is with a GET. This is not recommended, as you must pass all of your JSON in via a URL! This could be very big, very fast. However, it does work. So, write the following script.

```
#!/usr/bin/python3

import requests
import json

# define the URL we want to use
GETURL = "http://validate.jsontest.com/"

def main():
    # test data to validate as legal json
    mydata = {"fruit": ["apple", "pear"], "vegetable": ["carrot"]}

    ## the next two lines do the same thing
    ## we take python, convert to a string, then strip out whitespace
    #jsonToValidate = "json=" + str(mydata).replace(" ", "")
    #jsonToValidate = f"json={ str(mydata).replace(' ', '') }"
    ## slightly different thinking
    ## user json library to convert to legal json, then strip out whitespace
    jsonToValidate = f"json={ json.dumps(mydata).replace(' ', '') }"

    # use requests library to send an HTTP GET
    resp = requests.get(f"{GETURL}?{jsonToValidate}")

    # strip off JSON response
    # and convert to PYTHONIC LIST / DICT
    respjson = resp.json()

    # display our PYTHONIC data (LIST / DICT)
    print(respjson)

    # JUST display the value of "validate"
    print(f"Is your JSON valid? {respjson['validate']}")

if __name__ == "__main__":
    main()
```

9. Save and exit with `:wq`
10. Run your code.

```
student@bchd:~/mycode$ python3 jsontest/jsontestValidateGET.py
```

11. This should produce valid JSON. Which is cool, but let's try that with a POST. Unlike a GET, a POST allows for the attachment of a 'form', which can contain key=value pairs. The attachment can be of nearly any size, which makes it a bit more desirable than a GET. Additionally, we do not have white space concerns as we did when we jammed our JSON into a URL.

```
student@bchd:~/mycode$ vim jsontest/jsontestValidatePOST.py
```

12. Create the following script to validate with a POST. The trickiest thing about this exercise is that the documentation does a poor job explaining that the proper formatting for your form data is `{"json" : "the json to validate"}`.

```
#!/usr/bin/python3

import requests

# define the URL we want to use
POSTURL = "http://validate.jsontest.com/"

def main():
    # test data to validate as legal json
    # when a POST json= is replaced by the KEY "json"
    # the key "json" is mapped to a VALUE of the json to test
    # because the test item is a string, we can include whitespaces
    mydata = {"json": "{'fruit': ['apple', 'pear'], 'vegetable': ['carrot']}" }

    # use requests library to send an HTTP POST
    resp = requests.post(POSTURL, data=mydata)

    # strip off JSON response
    # and convert to PYTHONIC LIST / DICT
    respjson = resp.json()

    # display our PYTHONIC data (LIST / DICT)
    print(respjson)

    # JUST display the value of "validate"
    print(f"Is your JSON valid? {respjson['validate']}")

if __name__ == "__main__":
    main()
```

13. Save and exit with `:wq`
14. Run you code.

```
student@bchd:~/mycode$ python3 jsontest/jsontestValidatePOST.py
```

15. Okay. Now a bit of a challenge. Write a single script that utilizes the APIs on https://jsontest.com to perform the following:
    - **PART A** Pull timestamp of now (format is up to you)
    - **PART B** Pull the IP address of your current system
    - **PART C** Read in a list of servers from a file called, `myservers.txt` (you'll need to make this)
    - **PART D** format the data in the following manner: `{"json": "time: <<PART A>>, ip: <<PARTB>>, mysvrs: [ <<PARTC>> ]"}`
    - **PART E** Validate your JSON with a POST
16. The solution to the challenge above is below.
17. Create your hosts file, `myservers.txt`

```
student@bchd:~/mycode$ vim jsontest/myservers.txt
```

18. Enter the following into the file

```
host1
host2
host3
```

19. Save and exit with `:wq`
20. Create the solution script.

```
student@bchd:~/mycode$ vim jsontest/jsontestValidatePOST02.py
```

21. The following is one possible solution.

```
#!/usr/bin/python3

import requests

# define the URL we want to use
TIMEURL = "http://date.jsontest.com"
IPURL = "http://ip.jsontest.com"
VALIDURL = "http://validate.jsontest.com/"

def main():
    ## PART A
    ## pull a time object from date.jsontest.com
    # make the request
    resp = requests.get(TIMEURL)
    # pull json off 200 response
    # and change to PYTHONIC data
    mytime = resp.json()
    # pull out the value associated with the KEY "time"
    # then strip out all whitespaces
    # replace colons with hyphens
    mytime = mytime["time"].replace(" ", "").replace(":", "-")

    ## PART B
    ## make the request
    resp = requests.get(IPURL)
    myip = resp.json()
    print(myip)
    ## grab the value associated with the KEY "ip"
    myip = myip["ip"]

    ## PART C
    ## read a list of hosts out of a flat file
    with open("/home/student/mycode/jsontest/myservers.txt") as myfile:
        mysvrs = myfile.readlines()

    ## PART D
    # test data to validate as legal json
    # when a POST json= is replaced by the KEY "json"
    # the key "json" is mapped to a VALUE of the json to test
    # because the test item is a string, we can include whitespaces
    # format for requests to validate.testjson.com is...
    # data={"json": "json you want to validate as str"}
    jsonToTest = {}
    jsonToTest["time"] = mytime
    jsonToTest["ip"] = myip
    jsonToTest["mysvrs"] = mysvrs

    mydata = {}
    mydata["json"] = str(jsonToTest)

    ## PART E
    # use requests library to send an HTTP POST
    resp = requests.post(VALIDURL, data=mydata)

    # strip off JSON response
    # and convert to PYTHONIC LIST / DICT
    respjson = resp.json()

    # display our PYTHONIC data (LIST / DICT)
    print(respjson)

    # JUST display the value of "validate"
    print(f"Is your JSON valid? {respjson['validate']}")

if __name__ == "__main__":
    main()
```

Save and exit with `:wq`

22.
23. Run your solution.

```
student@bchd:~/mycode$ python3 jsontest/jsontestValidatePOST02.py
```

24. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "GET vs POST"`
    - `git push origin`

27. APIs and Dev Keys

Lab Objective

The objective of this lab is to create a Python client that can query APIs across a WAN. As a Python programmer, you'll be expected to work with APIs, which means understanding protocols like HTTP(S) and data structures like JSON. It doesn't matter if you're on the server or network side of automation, APIs are everywhere.

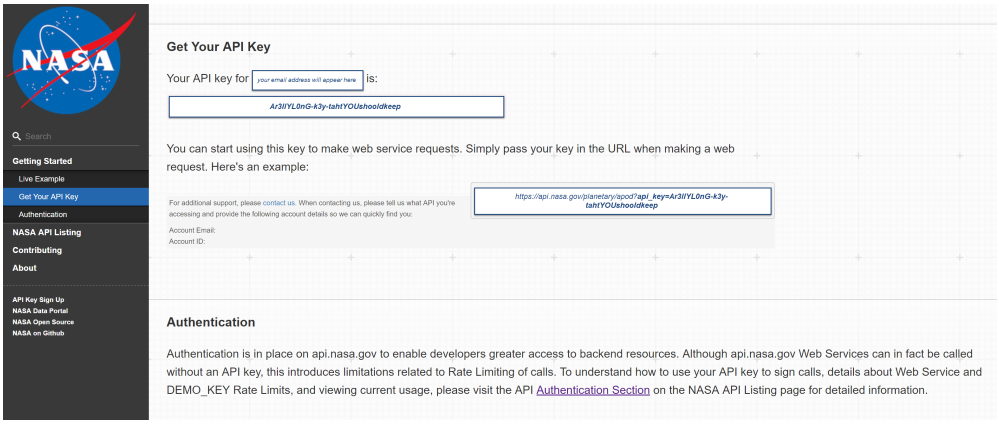
In addition to giving us things like Velcro and camera phones, NASA also will give us a massive amount of amazing data via RESTful APIs. In this lab and others we'll use NASA's free data to improve our skills writing Python programs and starting to make API calls across networks.

This lab is broken into **two** parts:

- **PART I** - RESTful API calls to the NASAs Picture of the Day(APOD) API.
- **PART II** - RESTful API calls to the Near Earth Object Web Service API.

Procedure

1. **PART I: RESTful API Request to the NASA APOD**
2. Open a new browser tab and visit NASA's Open API Data Set https://api.nasa.gov/
3. Apply for a API developer's key on NASA's website. It's free, and will entitle you to 1000 API requests per hour. The site to apply for a dev key is here: https://api.nasa.gov/
4. You'll need to supply your name and email address. Your dev key and account information will be emailed to you but you should record the key now. You'll need it when you make request to various NASA APIs. If you're a bit lost, see the screenshot below. You should be seeing something like it by the time you're at this step.



5. Now that you're armed with a NASA dev key, let's check out one of their most popular projects, NASA's Picture of the Day (APOD).
6. To explore this API. Let's start in the terminal. Type `python3`

```
student@bchd:~$ python3
```

7. First import the standard library for working with HTTP.

```
>>> import urllib.request
```

8. Define the NASA url.

```
>>> nasaurl = "https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY"
```

9. Now make a request using the NASA provided DEMO_KEY. We don't want to over use this option, and the number of requests per day is highly limited. But it should work for now.

```
>>> apodurlobj = urllib.request.urlopen(nasaurl)
```

10. Discover the names in the response object. Ignore anything with double-underscores around it. The other names are attributes or methods.

```
>>> dir(apodurlobj)
```

11. Show the response code attribute that is contained within our object.

```
>>> apodurlobj.code
```

12. Show the response HTTP message attribute that is contained within our object.

```
>>> apodurlobj.msg
```

13. Show the length (octets) contained within the response HTTP message.

```
>>> apodurlobj.length
```

14. Try displaying the attached json().

```
>>> apod = apodurlobj.read().decode("utf-8")
```

15. Some people use pretty print to make json more readable. Try that out. First `import pprint`

```
>>> import pprint
```

16. Try to pretty print the data structure.

```
>>> pprint.pprint(apod)
```

17. Okay, enough exploration. Try exiting.

```
>>> exit()
```

18. In a terminal, put your NASA API key in a location that will not get committed to GitHub. Your home directory is a good spot.

```
student@bchd:~$ vim ~/nasa.creds
```

19. Put your API key in this file. Don't put an 'extra' line feed.

20. Save and exit with `:wq`

21. Make a directory to work in.

```
student@bchd:~$ mkdir -p ~/mycode/nasa/
```

22. Move into the new directory.

```
student@bchd:~$ cd ~/mycode/nasa/
```

23. Use vim to write a script:

```
student@bchd:~/mycode/nasa$ vim apod.py
```

24. Create the following script. We can improve the design of this script later. First, let's just get something that works. The solution below uses the `urllib.request` and `json` libraries from the Python standard library.

```
#!/usr/bin/python3
import urllib.request
import json

## uncomment this import if you run in a GUI
## and want to open the URL in a browser
## import webbrowser

NASAAPI = "https://api.nasa.gov/planetary/apod?"

def main():
    ## Define creds
    with open("/home/student/nasa.creds") as mycreds:
        nasacreds = mycreds.read()

    ## remove any "extra" new line feeds on our key
    nasacreds = "api_key=" + nasacreds.strip("\n")

    ## Call the webservice with our key
    apodurlobj = urllib.request.urlopen(NASAAPI + nasacreds)

    ## read the file-like object
    apodread = apodurlobj.read()

    ## decode JSON to Python data structure
    apod = json.loads(apodread.decode("utf-8"))

    ## display our Pythonic data
    print("\n\nConverted Python data")
    print(apod)

    print()

    print(apod["title"] + "\n")

    print(apod["date"] + "\n")

    print(apod["explanation"] + "\n")

    print(apod["url"])

    ## Uncomment the code below if running in a GUI
    ## and you want to open the URL in a browser
    ## use Firefox to open the HTTPS URL
    ## input("\nPress Enter to open NASA Picture of the Day in Firefox")
    ## webbrowser.open(decodeapod["url"])
if __name__ == "__main__":
    main()
```

25. Save and exit with `:wq`

```
student@bchd:~/mycode/nasa$ python3 apod.py
```

26. Run your code. It should work, but we can do better by using the `requests` library.

```
student@bchd:~/mycode/nasa$ python3 -m pip install requests
```

27. Create a new script, `apod02.py`

```
student@bchd:~/mycode/nasa$ vim apod02.py
```

28. Create the following. In addition to the requests library, we'll also break our code to include an additional function, `returncreds()`. The function opens our `nasa.creds` file. One benefit of isolating code within functions, is that is more easily identifiable, reused, and debugged.

```
#!/usr/bin/python3

import requests

NASAAPI = "https://api.nasa.gov/planetary/apod?"

# this function grabs our credentials
def returncreds():
    ## first I want to grab my credentials
    with open("/home/student/nasa.creds", "r") as mycreds:
        nasacreds = mycreds.read()
    ## remove any newline characters from the api_key
    nasacreds = "api_key=" + nasacreds.strip("\n")
    return nasacreds

# this is our main function
def main():
    ## first grab credentials
    nasacreds = returncreds()

    ## make a call to NASAAPI with our key
    apodresp = requests.get(NASAAPI + nasacreds)

    ## strip off json
    apod = apodresp.json()

    print(apod)

    print()

    print(apod["title"] + "\n")

    print(apod["date"] + "\n")

    print(apod["explanation"])

    print(apod["url"])

if __name__ == "__main__":
    main()
```

29. Save and exit with `:wq`

```
student@bchd:~/mycode/nasa$ python3 apod02.py
```

30. **PART II: RESTful API Request to the NASA NEOWS**

31. In the second part of this lab, the data set we'll work is the **Near Earth Object Web Service**. With NeoWs a user can search for Asteroids based on their closest approach date to Earth, look up a specific asteroid with its NASA JPL small body id, and browse the overall data-set.

32. **EXAMPLE:** *Retrieve a list of Asteroids based on their closest approach date to Earth.*

```
GET https://api.nasa.gov/neo/rest/v1/feed?start_date=START_DATE&end_date=END_DATE&api_key=API_KEY
```

33. Open a new browser tab. You don't need to do this step within the remote desktop. If you haven't already, check out NASA's second most popular project, NASA Near Earth Object Web Service: https://api.nasa.gov/api.html#NeoWS

34. This part of the lab will also require your NASA developer key.

35. Open a new script with vim (keep working in the same directory as PART I).

```
student@bchd:~/mycode/nasa$ vim neows.py
```

36. Copy the following codeblock into your new script:

```
#!/usr/bin/python3
import requests

## Define NEOW URL
NEOURL = "https://api.nasa.gov/neo/rest/v1/feed?"

# this function grabs our credentials
# it is easily recycled from our previous script
def returncreds():
    ## first I want to grab my credentials
    with open("/home/student/nasa.creds", "r") as mycreds:
        nasacreds = mycreds.read()
    ## remove any newline characters from the api_key
    nasacreds = "api_key=" + nasacreds.strip("\n")
    return nasacreds

# this is our main function
def main():
    ## first grab credentials
    nasacreds = returncreds()

    ## update the date below, if you like
    startdate = "start_date=2019-11-11"

    ## the value below is not being used in this
    ## version of the script
    # enddate = "end_date=END_DATE"

    # make a request with the request library
    neowrequest = requests.get(NEOURL + startdate + "&" + nasacreds)

    # strip off json attachment from our response
    neodata = neowrequest.json()

    ## display NASAs NEOW data
    print(neodata)

if __name__ == "__main__":
    main()
```

37. Save and exit with `:wq`
38. Change your permission.

```
student@bchd:~/mycode/nasa$ chmod u+x neows.py
```

39. Run your code (the method below depends on having a *shebang* set as the first line of your code).

```
student@bchd:~/mycode/nasa$ ./neows.py
```

40. **CODE CUSTOMIZATION 01** - Looking for a bit of a challenge? Make the program accept a start date from a user, and an (optional) end parameter.
41. **CODE CUSTOMIZATION 02** - Try reading your API key in with the https://pypi.org/project/python-dotenv/ project. This makes it easy to interact with environmental variables from your local shell.
42. **CODE CUSTOMIZATION 03** - Find an additional NASA API listed on https://api.nasa.gov/ and write Python code to interact with it.
43. If you followed the labs, your NASA API key should **NOT** be anywhere in your current code. Remember, we saved it in a file `~/nasa.creds`, which is *not* being tracked by git. If the API key was coded into your solution, you should remove it before performing the following git operations.
44. If you're tracking your code in a SCM, issue the following commands:
    - `cd ~/mycode/`
    - `git add *`
    - `git commit -m "Working with NASA APIs"`
    - `git push origin`

28. RESTful APIs and Dev Keys

Lab Objective

The objective of this lab is to explore a vast data set as provided by the Marvel API. Marvel requires a developer's account to access their data but creating an account only requires a valid email address.

This lab suggests one of many applications that is possible given the 70 years of Marvel history. After building the Python application described in this lab you're welcome to be creative and build your own application using the Marvel API.

Remember, all API keys are a source of great power. If exchanged over HTTPS, you can be fairly certain that your credentials are not being exposed. If exchanged over HTTP, then you better be certain no sensitive data is being transmitted. In this lab, we'll learn how to hide our data in a secure fashion.

Just remember what Gandalf said to... Cyclops concerning API keys- 'Keep it secret, and keep it safe!'

Before you get started, read about what you're getting into: https://developer.marvel.com/

Procedure

1. Make your Marvel dev account. Head on over to https://developer.marvel.com/
2. Click your way through the creation process.
    1. Enter your birthday **CLICK**
    2. Enter your personal info (probably uncheck 'I want Marvel fan mail') **CLICK**
    3. Scroll to the bottom and check "Agree" **CLICK**
    4. To enable your API keys you receive, click the link in the confirmation email you receive. **OPEN EMAIL -> CLICK**
3. You get both a public and private key. Copy these both to a safe location. All calls to the Marvel Comics API must pass your public key via an `apikey` parameter. Please keep your private key private! Do not store your private key in publicly available code or repositories that are accessible to the public. Do not accidentally leave it at the bar.
4. The TLDR on this is as follows: server-side applications must pass two parameters in addition to the apikey parameter:
    - **ts** - a timestamp (or other long string which can change on a request-by-request basis)
    - **hash** - a md5 digest of the ts parameter, your private key and your public key (e.g. md5(ts+privateKey+publicKey)
5. For example, a user with a public key of "1234" and a private key of "abcd" could construct a valid call as follows: `http://gateway.marvel.com/v1/public/comics?ts=1&apikey=1234&hash=ffd275c5130566a2916217b101f26150`
    - The hash value is the md5 digest of 1abcd1234
6. The value **ts** is unique each time, which means we'll need to calculate a unique hash each time. We don't want to risk saving our private API key to the cloud but we do need it available to our script. Let's place it in a file in our home directory (which is encrypted).

    ```
    student@bchd:~$ vim marvel.priv
    ```
7. Copy and paste your **private key** from the Marvel website into this file.
8. Save and exit with `:wq`
9. Let's just put our public key in a file located in the home folder as well. That way if someone else wants to use our code they don't need to edit the code, just substitute in their own keyfiles.

    ```
    student@bchd:~$ vim marvel.pub
    ```
10. Copy and paste your **public key** from the Marvel website into this file.
11. Save and exit with `:wq`
12. Create a directory to work in.

    ```
    student@bchd:~$ mkdir mycode/xmen
    ```
13. Move into the new directory:

    ```
    student@bchd:~$ cd mycode/xmen/
    ```
14. Ensure you have the `requests` library installed. This is a third party library and is a replacement for `urllib.request`.

```
student@bchd:~/mycode/xmen$ python3 -m pip install requests
```

15. It seems bad practice as well as insecure and inflexible to hard code where our private API key is stored on our system, so let's pass this as an argument. We can do this with `argparse`. The `argparse` library has been available in the standard library since Python 3.2 and replaces `sys` as a more comprehensive way to collect command line arguments. Maybe one day the `requests` library will become part of the standard library as well.

16. Create a new script.

```
student@bchd:~/mycode/xmen$ vim marvel_api_01.py
```

17. Copy and paste the following into your script.

```
#!/usr/bin/env python3
"""Marvel Python Client
RZFeeser@alta3.com | Alta3 Research"""

# standard library imports
import argparse   # pull in arguments from CLI
import time       # create time stamps (for our RAND)
import hashlib    # create our md5 hash to pass to dev.marvel.com
from pprint import pprint # we only want pprint() from the package pprint

# 3rd party imports
import requests   # python3 -m pip install requests

## Define the API here
API = 'http://gateway.marvel.com/v1/public/characters'

## Calculate a hash to pass through to our MARVEL API call
## Marvel API wants md5 calc md5(ts+privateKey+publicKey)
def hashbuilder(rand, privkey, pubkey):
    return hashlib.md5((f"{rand}{privkey}{pubkey}").encode('utf-8')).hexdigest()  # create an MD5 hash of our identifers

## Perform a call to MARVEL Character API
## http://gateway.marvel.com/v1/public/characters
## ?name=Spider-Man&ts=1&apikey=1234&hash=ffd275c5130566a2916217b101f26150
def marvelcharcall(rand, keyhash, pubkey, lookmeup):
    r = requests.get(f"{API}?name={lookmeup}&ts={rand}&apikey={pubkey}&hash={keyhash}")  # send an HTTP GET to this location

    # the marvel APIs are "flakey" at best, so check for a 200 response
    if r.status_code != 200:
        response = None     #
    else:
        response = r.json()

    # return the HTTP response with the JSON removed
    return response


def main():

    ## harvest private key
    with open(args.dev) as pkey:
        privkey = pkey.read().rstrip('\n')

    ## harvest public key
    with open(args.pub) as pkey:
        pubkey = pkey.read().rstrip('\n')

    ## create an integer from a float timestamp (for our RAND)
    rand = str(time.time()).rstrip('.')

    ## build hash with hashbuilder(timestamp, privatekey, publickey)
    keyhash = hashbuilder(rand, privkey, pubkey)

    ## call the API with marvelcharcall(timestamp, hash, publickey, character)
    result = marvelcharcall(rand, keyhash, pubkey, "Wolverine")  # search for Wolverine

    ## display results
    pprint(result)

## Define arguments to collect
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    # This allows us to pass in public and private keys
    parser.add_argument('--dev', help='Provide the /path/to/file.priv containing Marvel private developer key')
    parser.add_argument('--pub', help='Provide the /path/to/file.pub containing Marvel public developer key')
```

```
    args = parser.parse_args()
    main()
```

18. Save and exit with `:wq`
19. Run your script.

```
student@bchd:~/mycode/xmen$ python3 marvel_api_01.py --dev /home/student/marvel.priv --pub /home/student/marvel.pub
```

20. You should get results about the character Wolverine (Logan). Our function call to `marvelcharcall` is hard-coded to look up this character.
21. Let's write a new script that allows us to look up any character we wish.

```
student@bchd:~/mycode/xmen$ vim marvel_api_02.py
```

22. Copy and paste the following into your script:

```
#!/usr/bin/env python3
"""Marvel Python Client
RZFeeser@alta3.com | Alta3 Research"""

# standard library imports
import argparse   # pull in arguments from CLI
import time       # create time stamps (for our RAND)
import hashlib    # create our md5 hash to pass to dev.marvel.com
from pprint import pprint # we only want pprint() from the package pprint

# 3rd party imports
import requests   # python3 -m pip install requests

## Define the API here
API = 'http://gateway.marvel.com/v1/public/characters'

## Calculate a hash to pass through to our MARVEL API call
## Marvel API wants md5 calc md5(ts+privateKey+publicKey)
def hashbuilder(rand, privkey, pubkey):
    return hashlib.md5((f"{rand}{privkey}{pubkey}").encode('utf-8')).hexdigest()  # create an MD5 hash of our identifers

## Perform a call to MARVEL Character API
## http://gateway.marvel.com/v1/public/characters
## ?name=Spider-Man&ts=1&apikey=1234&hash=ffd275c5130566a2916217b101f26150
def marvelcharcall(rand, keyhash, pubkey, lookmeup):
    r = requests.get(f"{API}?name={lookmeup}&ts={rand}&apikey={pubkey}&hash={keyhash}")  # send an HTTP GET to this location

    # the marvel APIs are "flakey" at best, so check for a 200 response
    if r.status_code != 200:
        response = None     #
    else:
        response = r.json()

    # return the HTTP response with the JSON removed
    return response


def main():

    ## harvest private key
    with open(args.dev) as pkey:
        privkey = pkey.read().rstrip('\n')

    ## harvest public key
    with open(args.pub) as pkey:
        pubkey = pkey.read().rstrip('\n')

    ## create an integer from a float timestamp (for our RAND)
    rand = str(time.time()).rstrip('.')

    ## build hash with hashbuilder(timestamp, privatekey, publickey)
    keyhash = hashbuilder(rand, privkey, pubkey)

    ## call the API with marvelcharcall(timestamp, hash, publickey, character)
    result = marvelcharcall(rand, keyhash, pubkey, args.hero)

    ## display results
    pprint(result)

## Define arguments to collect
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    # This allows us to pass in public and private keys
    parser.add_argument('--dev', help='Provide the /path/to/file.priv containing Marvel private developer key')
    parser.add_argument('--pub', help='Provide the /path/to/file.pub containing Marvel public developer key')

    ## This allows us to pass the lookup character
```

```
parser.add_argument('--hero', help='Character to search for within the Marvel universe')
args = parser.parse_args()
main()
```

23. Save and exit with `:wq`
24. Run your script.

```
student@bchd:~/mycode/xmen$ python3 marvel_api_02.py --dev /home/student/marvel.priv --pub /home/student/marvel.pub --hero Rogue
```

25. Try issuing a few more calls. Each time, change the hero you search for. If you're not a Marvel fan, try out `--hero Iceman`, or `--hero Archangel`. Also try out some false tests, `--hero Bicycle-Repairman` and `--hero Kalegirl` should fail.
26. Use your own mechanism to make the response more readable. This will likely mean *limiting* the amount of data that is actually displayed to the user (this is common practice).
27. **CODE CUSTOMIZATION 01 -** Use the Marvel Developer's page to research additional parameters that might be defined along with your API lookup to the Marvel character database.
28. **CODE CUSTOMIZATION 02 -** Create a mechanism to dig deeper into the returned data. If links are available, give the user options to automatically open them within the browser.
29. Great job! That's it for this lab. If you make something worth sharing, be sure to let the instructor know.
30. If you're tracking in an SCM, and you'd like to backup your code, run the following commands:
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "All about Xaviers School for Gifted Youngsters"`
    - `git push origin`

29. Lecture - OAuth

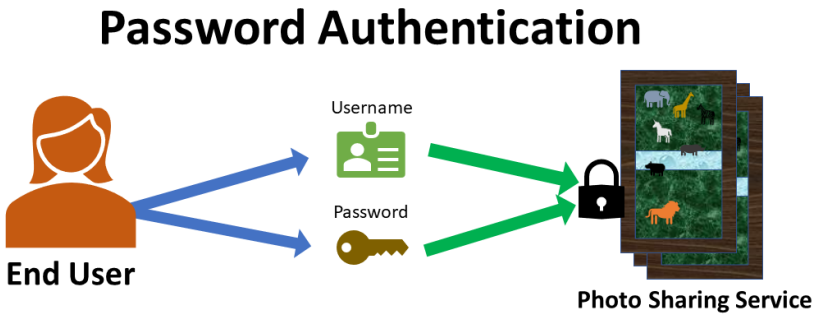
Lab Objective

The objective of this lab is to design an API that is able to request and use an OAuth token as its auth method for performing specific actions. OAuth is an **Authorization Framework** version 2.0 (current). It enables third-party applications to obtain limited access to an HTTP service, such as a website. This is most typically done via an **End User** approving an interaction between the HTTP service and the third-party application.

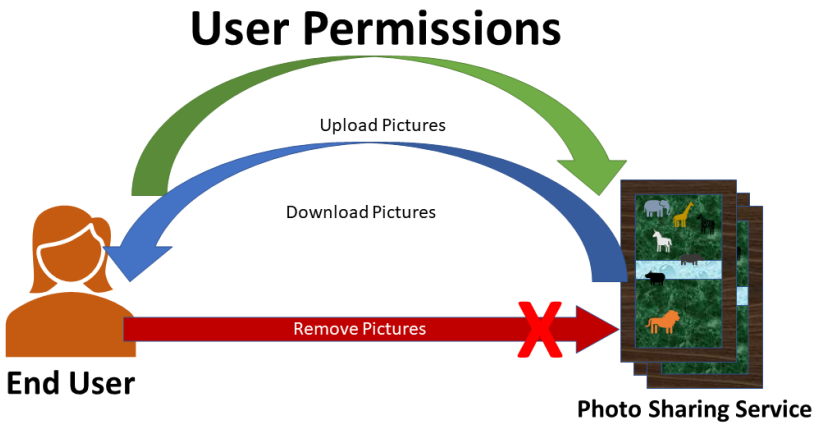
You will achieve this by generating an **OAuth token** token on GitHub and used it to authenticate using our own **3rd Party Service (python script)** with our **Resource Server (GitHub)** and perform the actions of listing and creating repositories.

Photo Printing Example - OAuth Application

Let's imagine that an **End User** has recently taken a trip to Wakanda, and they stored all of their wonderful pictures of their journey on an online photo storage and sharing service, more aptly known as a **Resource Server**. In order to access this service, the **End User** must authenticate using a *username and password*, like so:

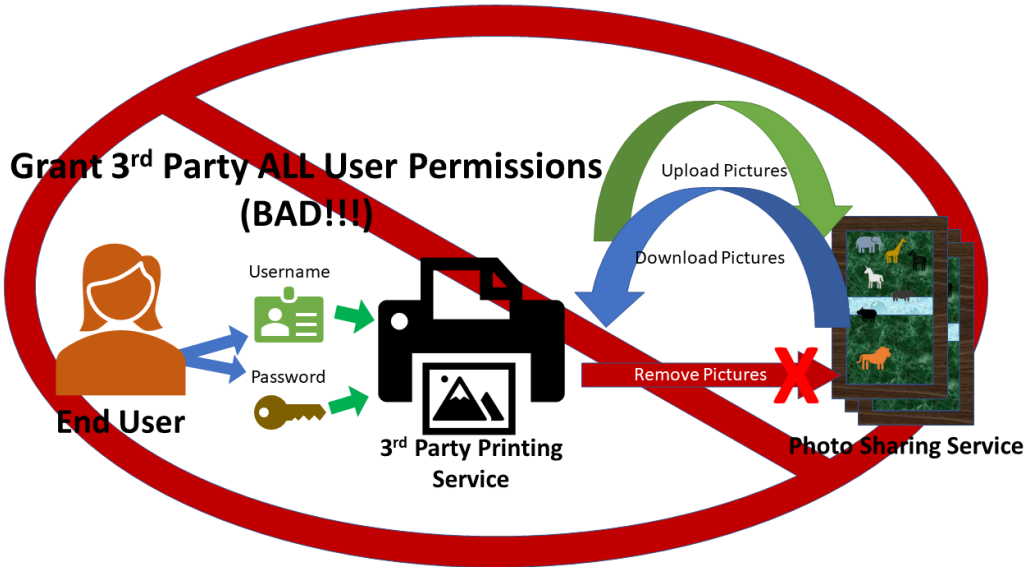


Once this user has authenticated with their **Resource Server**, they will be able to perform **ALL** of the actions available to their specific user. In this case, let's imagine that they are able to upload, download, and delete pictures from this Photo Sharing Service.



However, they have a problem. In order to afford the trip to Wakanda, they had to sell their own personal printer, and all of it's ink. They cannot print their awesome pictures at home, and now must have some other service print these photos, referred to as a **Client**, or a **3rd Party Service**. But they don't want to go through the hassle of downloading these super-hi-resolution photos, and then uploading them to the printing service. They want the internet to take care of this hassle. So they now face this conundrum:

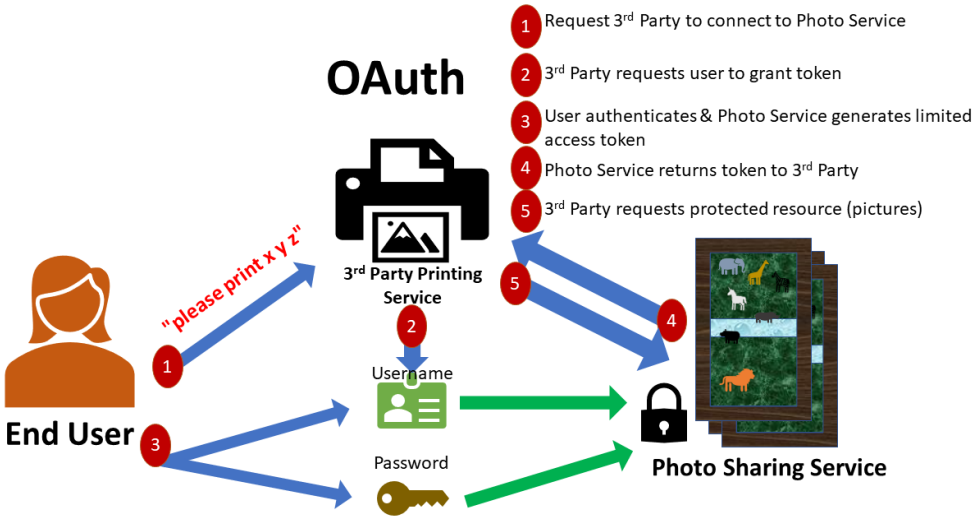
Option 1 is to simply give this **3rd Party Service** their login information to the Photo Sharing Service. This would let the **3rd Party Service** access the photos as requested. However, this also will grant them **ALL OF THE PERMISSIONS THE END USER HAS**. This is not a good plan.



Option 2 is a much better option, known as **OAuth**. This process is not simple to set up on the backend, but makes the authorization flow for users incredibly simple, and it will allow for the **3rd Party Service** to have a **limited scope** of actions to take in your **Resource Server**. You may have even already used **OAuth** without knowing it.

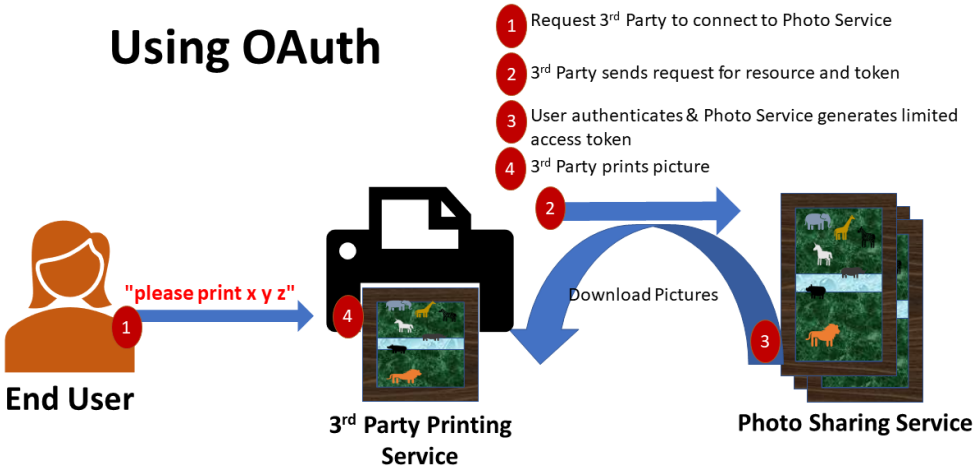
From the user's point of view, the flow in the diagram below would be: 0. Sign in to the **3rd Party Printing Service**.

1. Ask the **3rd Party Printing Service** to connect to your **Photo Sharing Service**.
2. Get Prompted with a "Sign in" page for your **Photo Sharing Service**.
3. Sign in to the **Photo Sharing Service** and click the button that says "Allow **3rd Party Printing Service** to access your Photos."
4. Use your **3rd Party Printing Service** to select photos and order them to be printed



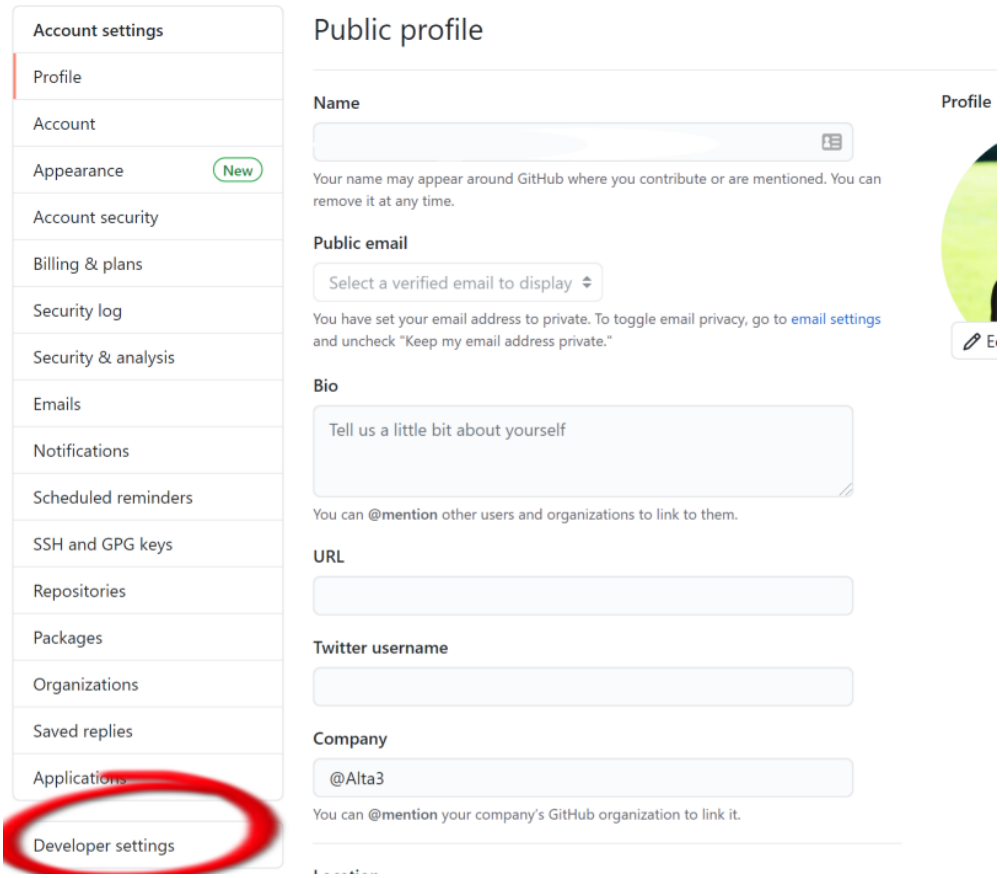
Now that the first interaction has between your **3rd Party Printing Service** and the **Photo Sharing Service** has been authorized by you, there is no further need for the user to authenticate the **3rd Party Printing Service**. It is the responsibility of the **3rd Party Printing Service** to retain and periodically update the token that it must use to authenticate to the **Photo Sharing Service**.

Additionally, the **Photo Sharing Service** has the responsibility to assure that the token that does get generated offers only limited actions to the service using it. In this case, there is no reason why a printing service would ever need to add or delete pictures from your account, so the only permission that should be granted to it would be to browse and download these images.

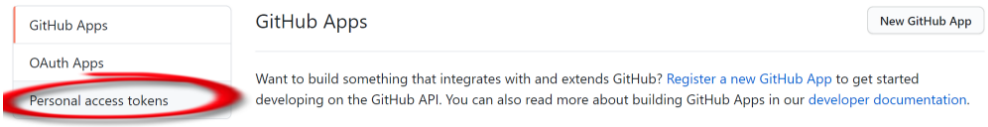


Procedure

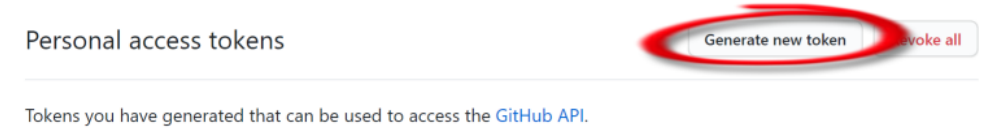
1. Using your local browser, log in to https://github.com.
2. Click on your profile picture in the top right corner, then select **Settings** (near the bottom of the dropdown).
3. On the left hand navigation bar, near the bottom, click on **Developer Settings**.



4. Next, select **Personal access tokens**.

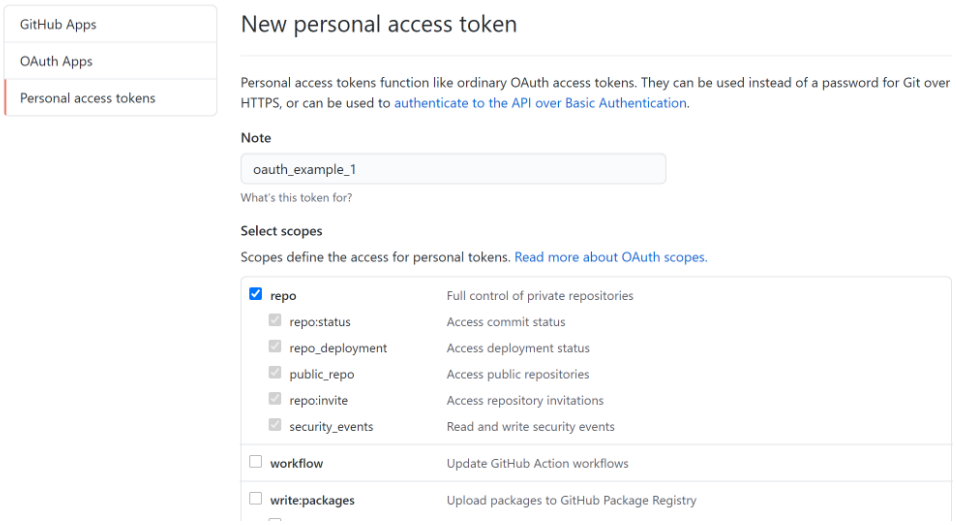


5. Then **Generate new token**.



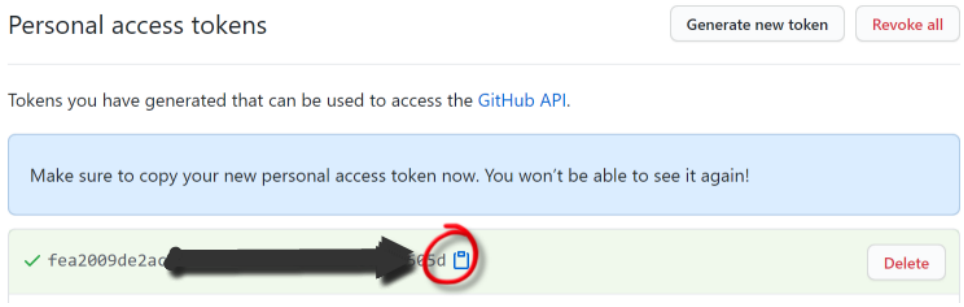
6. Now we need to give our personal access token a name, as well as one or more scopes. The name is up to you, or feel free to use the example given of **ouath_example_1**. Then, select the radio (checkmark) button to the left of the bold word **repo** inside of the **Select scopes** section of this page. This grants **ANYBODY WITH THIS TOKEN** complete access to your repos. All of them.

> For your token --- "Keep it secret. Keep it safe" - J. R. R. Tolkien



Scroll down to the bottom of the page and select the green **Generate Token** button.

7. You now should be back at your main **Personal access tokens** page, and see that there is a new token there for you. Make sure that you copy this token right now, and put it somewhere safe. As soon as you move off of this page, you lose the ability to view it any more. We will need this token for the rest of the lab.



8. **BACK IN YOUR TERMINAL ENVIRONMENT** you will soon be creating a new python script file for generating requests using your token. But first we will need to save this token in a safe location. Create the following file, and paste your token in there.

```
student@bchd:~$ vim ~/token

fea2009de2a192kdk9k10a3k39as605d
```

Save and Quit

9. Now we will make a new **oauth** directory and move into there.

```
student@bchd:~$ mkdir -p ~/mycode/oauth
```

10. Move into `~/mycode/oauth`

```
student@bchd:~$ cd ~/mycode/oauth
```

11. Before we get too far ahead of ourselves, let's take a look at what we can do with our token manually. The primary purpose that we tasked our token with doing is allowing us to connect to our private repos. But it also is a viable way to let us authenticate with GitHub's API for other use cases too. So before reaching into the realm of repos, let's try to do something simple like grab an SSH public key from some user's GitHub account. But even before this, perform the following command:

```
student@bchd:~/mycode/oauth$ curl https://api.github.com/rate_limit
```

```
{
  "resources": {
    "core": {
      "limit": 60,
      "remaining": 60,
      "reset": 1615234282,
      "used": 0
    },
    "graphql": {
      "limit": 0,
      "remaining": 0,
      "reset": 1615237644,
      "used": 0
    },
    "integration_manifest": {
      "limit": 5000,
      "remaining": 5000,
      "reset": 1615237644,
      "used": 0
    },
    "search": {
      "limit": 10,
      "remaining": 10,
      "reset": 1615234104,
      "used": 0
    }
  },
  "rate": {
    "limit": 60,
    "remaining": 60,
    "reset": 1615234282,
    "used": 0
  }
}
```


> This command shows us the number of requests that we have remaining for our IP address. There is a chance that you actually are out of any **remaining** due to others in this same IP address using them all up. That is okay, because this is currently being done without any authentication.

12. Next, we can use our token as an authentication method for GitHub's API. This will allow us to authenticate via our specific user, not just from a specific IP address (that you share with all of your classmates). Paste your personal token in where it says `<token>`. And notice how you have gone from a rate of 60 requests to 5000.

```
student@bchd:~/mycode/oauth$ curl -H "Authorization: token <token>" https://api.github.com/rate_limit
```

```
{
  "resources": {
    "core": {
      "limit": 5000,
      "used": 0,
      "remaining": 5000,
      "reset": 1615235630
    },
    "search": {
      "limit": 30,
      "used": 0,
      "remaining": 30,
      "reset": 1615234829
    },
    "graphql": {
      "limit": 5000,
      "used": 0,
      "remaining": 5000,
      "reset": 1615238369
    },
    "integration_manifest": {
      "limit": 5000,
      "used": 0,
      "remaining": 5000,
      "reset": 1615238369
    },
    "source_import": {
      "limit": 100,
      "used": 0,
      "remaining": 100,
      "reset": 1615234829
    },
    "code_scanning_upload": {
      "limit": 500,
      "used": 0,
      "remaining": 500,
      "reset": 1615238369
    }
  },
  "rate": {
    "limit": 5000,
    "used": 0,
    "remaining": 5000,
    "reset": 1615235630
  }
}
```


13. Let's try to use up some of those limits and see how they respond! To start with, we will do the next command four times unauthenticated, then check on the rate remaining. Feel free to put your own username in place of `JasonTrespel`, that is just this lab author's GitHub username.

```
student@bchd:~/mycode/oauth$ curl https://api.github.com/users/JasonTrespel/keys
```

Run this command four times.

14. Now, let's check to see how many requests you have remaining.

```
student@bchd:~/mycode/oauth$ curl https://api.github.com/rate_limit

...
  "rate": {
    "limit": 60,
    "remaining": 56,
    "reset": 1615234282,
    "used": 4
  }
}
```

15. Next, let's try to run the exact same curl command, except this time we will pass the Header for **Authorization** to the request, using our token. This time run this command seven times. Remember to paste your personal token in where it says `<token>`, and feel free to use your own username.

```
student@bchd:~/mycode/oauth$ curl -H "Authorization: token <token>" https://api.github.com/users/JasonTrespel/keys
```

> Run this command seven times.

16. Now we can see how many **Authenticated** requests we have left. Paste your personal token in where it says `<token>`.

```
student@bchd:~/mycode/oauth$ curl -H "Authorization: token <token>" https://api.github.com/rate_limit

...
  "rate": {
    "limit": 5000,
    "used": 7,
    "remaining": 4993,
    "reset": 1615235630
  }
}
```

17. Now that we have seen how to use **cURL** to authenticate with GitHub, let's attempt to do that with Python. The goal of these next several steps is to create a Python script that will let us list existing and create new repos on GitHub. Create the following file:

```
student@bchd:~/mycode/oauth$ vim repo_manager.py
```

18. Create the following script:

```
#!/usr/bin/env python3
"""GitHub Client - OAUth and API study | Alta3 Research"""

import json
import requests

MY_USERNAME = ""  # Make sure you put your own username in here!

def create_repo(repo_name: str, token: str) -> str:
    """
    This will create a repo for your GitHub account.
    """
    repo_data = {"name": repo_name}
    json_data = json.dumps(repo_data)
    headers = {"Authorization": f"token {token}"}
    r = requests.post(f"https://api.github.com/user/repos", data=json_data, headers=headers)
    response_code = r.status_code
    response = r.text
    print(response_code, response)
    return response

def show_repos(username: str, token: str) -> list:
    """
    This will list out all of the repos associated with your GitHub account.
    """
    url = f"https://api.github.com/users/{username}/repos"
    headers = {"Authorization": f"token {token}"}
    r = requests.get(url, headers=headers)
    resp_headers = r.headers
    print(f"""Rate - \n            Limit: {resp_headers['X-RateLimit-Limit']}
            Used: {resp_headers['X-RateLimit-Used']}
            Remaining: {resp_headers['X-RateLimit-Remaining']}""")
    repos = list(r.json())
    for repo in repos:
        print(repo['name'])
    return repos

def get_token() -> str:
    # Read token in from file
    with open("/home/student/token") as f:
        token = f.read().rstrip("\n")
        return token

if __name__ == "__main__":
    tkn = get_token()
    show_repos(MY_USERNAME, tkn)
    create_repo("learning_oauth", tkn)
```

> Make sure that you edit line 7 to include **YOUR GITHUB USERNAME!!** Also, if you do not have the **requests** package installed, install it now `python3 -m pip install requests`.

19. Now we can run our **repo_manager.py** script. This will list all of our GitHub repos, then will create a new GitHub repo named **learning_oauth** for us.

```
student@bchd:~/mycode/oauth$ python3 repo_manager.py

Rate -
        Limit: 5000
        Used: 1
        Remaining: 4999
mycode-dev
```

20. Hooray! We just used our **OAuth token** to authenticate using our own **3rd Party Service (python script)** with our **Resource Server (GitHub)** and perform the actions of listing and creating repositories!

21. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`

- `git commit -m "Exploring how to work with large data sets"`
- `git push origin`

Additional Reading

RFC 6749

- The OAuth2.0 Authorization Framework

30. Simple Object Access Protocol (SOAP) and Python

Lab Objective

The objective of this lab is to learn to to use Python to make requests to Simple Object Access Protocol (SOAP) APIs. Standardized by Microsoft, it uses XML in place of JSON for data exchange. While this is not at odds with RESTful APIs, it isn't what we typically see as the attachment on RESTful APIs.

Resources:

- ZEEP (Python SOAP client) https://docs.python-zeep.org/en/master/

Procedure

1. Create a new space in which to work.

```
student@bchd:~$ mkdir -p ~/mycode/soap/
```

2. Move into the directory.

```
student@bchd:~$ cd ~/mycode/soap/
```

3. Create a new script.

```
student@bchd:~/mycode/soap$ vim soap01.py
```

4. Take a peek at the SOAP APIs we are using http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso

5. Copy and paste the following into your script:

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   Using the requests library to service SOAP APIs"""

# python3 -m pip install requests
import requests

def main():
    # SOAP request URL
    url = "http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso"

    # structured XML
    payload = """<?xml version=\"1.0\" encoding=\"utf-8\"?>
                <soap:Envelope xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">
                    <soap:Body>
                        <CountryIntPhoneCode xmlns=\"http://www.oorsprong.org/websamples.countryinfo\">
                            <sCountryISOCode>IN</sCountryISOCode>
                        </CountryIntPhoneCode>
                    </soap:Body>
                </soap:Envelope>"""
    # headers
    headers = {
        'Content-Type': 'text/xml; charset=utf-8'
    }
    # POST request
    response = requests.request("POST", url, headers=headers, data=payload)

    # print the response
    print(response.text)
    print(response)

if __name__ == "__main__":
    main()
```

6. Save and exit with `:wq`

7. Great! Run your script.

```
student@bchd:~/mycode/soap$ python3 soap01.py
```

8. The result is an XML output. A description of this SOAP endpoint is found here http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso?op=CountryName

9. Install the `zeep` client.

```
student@bchd:~/mycode/soap$ python3 -m pip install zeep
```

10. Create a script that uses the zeep client.

```
student@bchd:~/mycode/soap$ vim soap02.py
```

11. Create the following script:

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   Using the zeep library to service SOAP APIs"""

# python3 -m pip install zeep
import zeep

# set the WSDL URL
wsdl_url = "http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso?WSDL"

# set method URL
method_url = "http://webservices.oorsprong.org/websamples.countryinfo/CountryIntPhoneCode"

# set service URL
service_url = "http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso"

# create the header element
header = zeep.xsd.Element(
    "Header",
    zeep.xsd.ComplexType(
        [
            zeep.xsd.Element(
                "{http://www.w3.org/2005/08/addressing}Action", zeep.xsd.String()
            ),
            zeep.xsd.Element(
                "{http://www.w3.org/2005/08/addressing}To", zeep.xsd.String()
            ),
        ]
    ),
)
# set the header value from header element
header_value = header(Action=method_url, To=service_url)

# initialize zeep client
client = zeep.Client(wsdl=wsdl_url)

# set country code for United States
country_code = "US"

# make the service call
result = client.service.CountryIntPhoneCode(
    sCountryISOCode=country_code,
    _soapheaders=[header_value]
)

# print the result
print(f"Phone Code for {country_code} is {result}")



# set country code for Japan
country_code = "JP"

# make the service call
result = client.service.CountryIntPhoneCode(
    sCountryISOCode=country_code,
    _soapheaders=[header_value]
)

# print the result
print(f"Phone Code for {country_code} is {result}")
print(result)
```

12. Save and exit with `:wq`
13. Great! Run your script.

```
student@bchd:~/mycode/soap$ python3 soap02.py
```

14. The script should execute without error. Answer the following questions:
    - **Q: What is the WSDL URL?**
        - *A: This is the service URL. To retrive it, visit http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso and click on the link "Service Description" at the top.*
    - **Q: What is the purpose of method_url and service_url?**
        - *A: These create the header element for the SOAP request.*
    - **Q: What does zeep do?**
        - *A: It abstracted the SOAP lookup. Once initialized, we can simply call the zeep service, client.service.CountryIntPhoneCode(), and pass it the parameters country_code and _soapheaders as a list*
15. **CHALLENGE (OPTIONAL)** - Find some additional country codes by visiting http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso/ListOfCountryNamesByCode/ then rewrite `soap02.py` to make look ups to these new countries.
16. If you're tracking your code, run the following commands:
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "python and soap"`
    - `git push origin`

31. Construct a SimpleHTTPServer and HTTP Client

Lab Objective

The objective of this lab is to use the standard library to construct an HTTP server and client. The server side will expose any files from the directory it is launched *from*. The client side will behave much like `urllib.request` and `requests`, in that we can send an HTTP message.

Resources:

- Python Standard Library - http.server
- Python Standard Library - http.client

Procedure

1. Start in the home directory.

```
student@bchd:~$ cd
```

2. Create a space to work in directory:

```
student@bchd:~$ mkdir -p ~/mycode/httpwork/
```

3. Our first goal is to launch an HTTP server. Create a script:

```
student@bchd:~$ vim ~/mycode/httpwork/custom_server.py
```

4. Create a script that will launch and HTTP server we can query against.

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   Using http.server to create a simple HTTP server."""

# standard library
import http.server
import socketserver

def main():
    """run-time code"""

    ## port to run on
    port = 9021

    ## handler is how to respond to an event
    ## any incoming HTTP message to the root "/"
    ## will provoke a 200+HTML describing files in the directory
    ## the server was launched from
    handler = http.server.SimpleHTTPRequestHandler

    ## build a server object that listens on all interfaces, described port and event handler
    httpd = socketserver.TCPServer(("", port), handler)

    print(("serving at port", port))

    ## start the server until it is interrupted
    httpd.serve_forever()

if __name__ == "__main__":
    main()
```

5. Press **Esc** and then save and exit with `:wq`

6. Notice that we could start our server on any port we wanted? In the code above, we choose port 9021.

7. Don't start the server yet. Instead, create a new script.

```
student@bchd:~$ vim ~/mycode/httpwork/custom_client.py
```

8. Create the following script.

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   Using http.client to create a simple HTTP client."""

import http.client

def main():

    ## think of this as setting up the connection
    conn = http.client.HTTPConnection("localhost", 9021)

    ## Send an HTTP request and store the HTTP response
    ##    from our webserver
    conn.request('HEAD', '/')

    ## Returns just the response that has been associated with
    ##    the **conn** object.
    res = conn.getresponse()

    ## response status and the reason to the screen.
    print(res.status, res.reason)

if __name__ == "__main__":
    main()
```

9. Press **Esc** and then save and exit with `:wq`
10. Move into the `~/mycode` directory.

```
student@bchd:~$ cd ~/mycode
```

11. Now run your server code.

```
student@bchd:~/mycode$ python3 ~/mycode/httpwork/custom_server.py
```

12. Split your screen with tmux (**ctrl** + `b` and then `"`)
13. Curl the server.

```
student@bchd:~$ curl http://127.0.0.1:9021
```

14. You should see a **Directory listing for /** of the files from within **/home/student/mycode/**, where we started the webservice from.
15. Leave the webserver running and execute your client.

```
student@bchd:~$ python3 ~/mycode/httpwork/custom_client.py
```

16. Looks like we got a **200 OK** message back from our HTTP server. Back to the client script.

```
student@bchd:~$ vim ~/mycode/httpwork/custom_client02.py
```

17. Let's add some code below the lines we currently have. We'll try making a GET message, which should return the data associated with the root directory we're probing on (which should be listing within the directory).

```
#!/usr/bin/env python3
"""Alta3 Research | rzfeeser@alta3.com
   (improved) http.client to create a simple HTTP client."""

import http.client

def main():

    ## think of this as setting up the connection
    conn = http.client.HTTPConnection("localhost", 9021)

    ## Send an HTTP request and store the HTTP response
    ##    from our webserver
    conn.request('HEAD', '/')

    ## Returns just the response that has been associated with
    ##    the **conn** object.
    res = conn.getresponse()

    ## response status and the reason to the screen.
    print(res.status, res.reason)

    ## this time we'll issue GET
    conn.request('GET', '/')

    ## res is equal to the response associated with conn
    res = conn.getresponse()

    ## print the response status code and reason
    print(res.status, res.reason)

    ## page_data is all of the data associated with res
    page_data = res.read()

    ## this will point out all of the data associated with res
    print(page_data)

if __name__ == "__main__":
    main()
```

18. Press **Esc** and then save and exit with `:wq`
19. Execute your second client script.

```
student@bchd:~$ python3 ~/mycode/httpwork/custom_client02.py
```

20. Cool! Looks like we got a **200 OK** message from the head message and a **200 OK** message from the get along with some **HTML** page data.
21. Close your second terminal by typing `exit`

```
student@bchd:~$ exit
```

22. Stop the simple HTTP server with **CTRL** + **c**
23. **CHALLENGE 01 (OPTIONAL)** - Upgrade the web client so that a user can indicate the type of HTTP request they wish to send to the server.
24. **CHALLENGE 02 (OPTIONAL)** - Upgrade the web client so that the data returned by the HTTP GET to a file called `http_readin.txt`
25. If you are tracking your code in an SCM, issue the following commands:

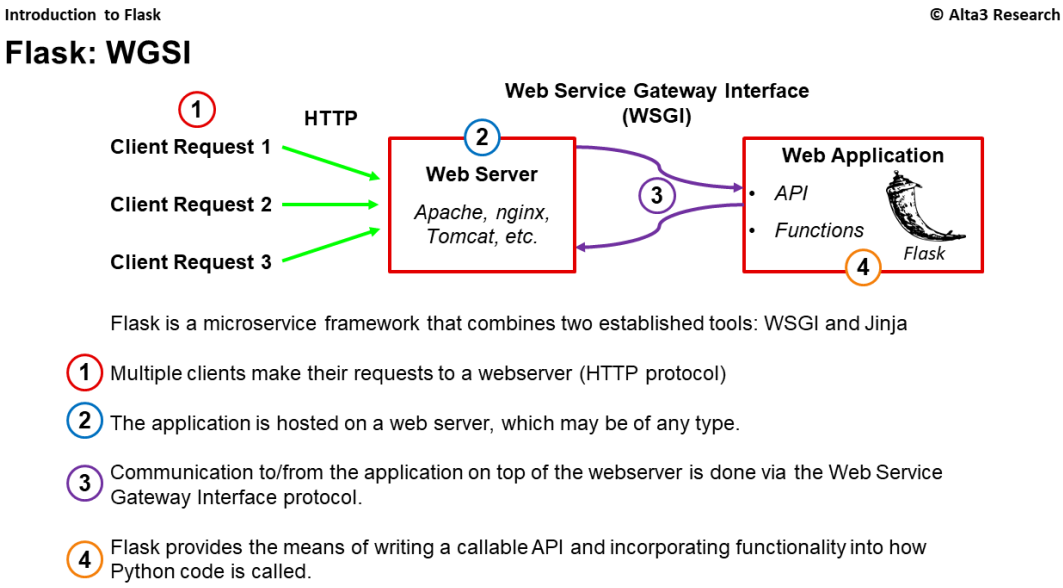
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "Building a simple Python HTTP server"`
    - `git push origin`

32. Lecture - Introduction to Flask

Lab Objective

The objective of this lab is to introduce Flask framework.

Procedures



Flask is a microservice framework that combines two established tools: WSGI and Jinja

1. Multiple clients make their requests to a webserver (HTTP protocol)
2. The application is hosted on a web server, which may be of any type.
3. Communication to/from the application on top of the webserver is done via the Web Service Gateway Interface protocol.
4. Flask provides the means of writing a callable API and incorporating functionality into how Python code is called.

33. Building APIs with Python

Lab Objective

The objective of this lab is to explore building APIs with the minimalist framework, Flask. Before we start, review the following terms:

- **Web Application Framework (Web Framework)** - A collection of libraries and modules that enables a web application developer to write applications without having to bother about low-level details such as protocols, thread management, etc.
- **Flask** - Web application framework written in Python. It is developed by Armin Ronacher, who leads an international group of Python enthusiasts named Pocco. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine. Both are Pocco projects. Flask is often referred to as a micro framework. It aims to keep the core of an application simple yet extensible. Flask does not have a built-in abstraction layer for database handling, nor does it have a form of validation support. Instead, Flask supports the extensions to add such functionality to the application.
- **Web Server Gateway Interface (WSGI)** - Adopted as a standard for Python web application development, WSGI is a specification for a universal interface between the web server and the web applications.
- **Werkzeug** - WSGI toolkit, implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.
- **Jinja2** - A popular templating engine for Python. A web templating system combines a template with a certain data source to render dynamic web pages.

Resources

- Python Flask Homepage

Procedure

1. Install the software library "Flask", which we will need for this script. This will allow us to create APIs.

```
student@bchd:~$ python3 -m pip install flask
```

2. Create a new directory to work in, `/home/student/mycode/flaskapi/`

```
student@bchd:~$ mkdir ~/mycode/flaskapi/
```

3. Move into the new directory.

```
student@bchd:~$ cd ~/mycode/flaskapi/
```

4. Create a new script, `myflask01.py`

```
student@bchd:~/mycode/flaskapi$ vim ~/mycode/flaskapi/myflask01.py
```

5. Copy and paste the following into your script:

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   A simple Flask server. Responds to HTTP 'GET /' requests
   with a 'Hello World' attached to a 200 response"""


# An object of Flask class is our WSGI application
from flask import Flask

# Flask constructor takes the name of current
# module (__name__) as argument
app = Flask(__name__)

# route() function of the Flask class is a
# decorator, tells the application which URL
# should call the associated function
@app.route("/")
def hello_world():
   return "Hello World"

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224) # runs the application
   # app.run(host="0.0.0.0", port=2224, debug=True) # DEBUG MODE
```

6. Save and exit with `:wq`

7. Run the script `myflask01.py`

```
student@bchd:~/mycode/flaskapi$ python3 myflask01.py
```

8. Along the right hand side of the screen, choose `aux1` in your drop down selector.
9. The `aux1` connection is a view of what is happening at `http://0.0.0.0:2224/`. Therefore, the `Hello World` should be returned.
10. From the drop down selector, choose `tmux` to return to your environment again.
11. Stop your Flask app by clicking on the terminal window it is running in and pressing **ctrl** + `c`
12. Modern web frameworks use the routing technique to help a user remember application URLs. It is useful to access the desired page directly without having to navigate from the home page. The route() decorator in Flask is used to bind URL to a function. For example, consider the following:

```
@app.route("/hello")
def hello_world():
    return "hello world"
```

13. Above, the URL `/hello` rule is bound to the `hello_world()` function. As a result, if a user visits the `http://0.0.0.0:2224/hello` URL, the output of the `hello_world()` function will be rendered in the browser.
14. The `add_url_rule()` function of an application object is also available to bind a URL with a function. As in the above example, `route()` is used. A decorator's purpose is also served by the following representation (again, just for your reading pleasure):

```
def hello_world():
   return "hello world"
app.add_url_rule("/hello", "hello", hello_world)
```

15. Create a new script, `myflask02.py`

```
student@bchd:~/mycode/flaskapi$ vim myflask02.py
```

16. It is possible to build a URL dynamically by adding variable parts to the rule parameter. This variable part is marked as `<variable-name>`. It is passed as a keyword argument to the function with which the rule is associated. In your next script, the rule parameter of route() decorator contains `<name>` variable part attached to URL `/hello`. Hence, if the http://localhost:5000/hello/Zuul is entered as a URL in the browser, `Zuul` will be supplied to `hello()` function as argument.
17. Copy and paste the following into `myflask02.py`

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   A simple Flask server. Responds to HTTP 'GET /hello/<name>' requests
   with 'Hello <name>' attached to a 200, where <name> is the same as what the
   requester sent to the endpoint."""

from flask import Flask
app = Flask(__name__)

@app.route("/hello/<name>")
def hello_name(name):
    return f"Hello {name}"
    ## V2 STYLE STRING FORMATTER - return "Hello {}".format(name)
    ## OLD STYLE STRING FORMATTER - return "Hello %s!" % name

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224) # runs the application
```

18. Press **Esc** and then save and exit with `:wq`
19. Run your script, `myflask02.py`

```
student@bchd:~/mycode/flaskapi$ python3 myflask02.py
```

20. *Once again,* along the right hand side of the screen, choose `aux1` in your drop down selector.
21. The current url is showing what is running at `https://0.0.0.0:2224/`, therefore, append your current URL with `hello/Worf%20Son%20of%20Mogh`
22. The `Hello Worf Son of Mogh` should be returned. Notice that we used ASCII characters, `%20` to indicate blank spaces.
23. From the drop down selector, choose `tmux` to return to your environment again.
24. Stop your Flask app by clicking on the terminal window it is running in and pressing **ctrl** + `c`
25. The `url_for()` function is very useful for dynamically building a URL for a specific function. The function accepts the name of a function as first argument and one or more keyword arguments, each corresponding to the variable part of URL. Write a script to demonstrate the `url_for()` function.

```
student@bchd:~/mycode/flaskapi$ vim myflask03.py
```

26. Copy and paste the following into `myflask03.py`

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   Exploring redirection with a simple Flask server. This server has
   the following endpoints:

   /admin            - returns 200 + 'Hello Admin'
   /guest/<guesty>   - returns 200 + 'Hello {guesty} Guest'
   /user/<name>      - returns 302 to one of the other 2 endpoints depending on the
                       <name> provided."""

# python3 -m pip install flask
from flask import Flask
from flask import redirect
from flask import url_for

# create flask app instance
app = Flask(__name__)

@app.route("/admin")
def hello_admin():
    return "Hello Admin"

@app.route("/guest/<guesty>")
def hello_guest(guesty):
    return f"Hello {guesty} Guest"
    #V2 FORMATTER - return "Hello {} Guest".format(guesty)
    #OLD FORMATTER - return "Hello %s as Guest" % guesty

@app.route("/user/<name>")
def hello_user(name):
    ## if you go to hello_user with a value of admin
    if name =="admin":
        # return a 302 response to redirect to /admin
        return redirect(url_for("hello_admin"))
    else:
        # return a 302 response to redirect to /guest/<guesty>
        return redirect(url_for("hello_guest",guesty = name))

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224) # runs the application
```

27. Press **Esc** and then save and exit with `:wq`

28. Run the script `myflask03.py`.

```
student@bchd:~/mycode/flaskapi$ python3 myflask03.py
```

29. The above script has a function user(name) which accepts a value to its argument from the URL. The `user()` function checks if an argument received matches `admin` or not. If it matches, the application is redirected to the `hello_admin()` function using `url_for()`. If there is no match, redirection will be to the `hello_guest()` function along with the received argument parameter passed to it.

30. Split the screen by pressing **ctrl** + `b` and then **SHIFT** + `"`

31. In the new split screen, try using CURL to access our endpoints. CURL is short for 'see-URL', and allows us to access HTTP resources from the CLI. It likely is installed, but it won't hurt to double-check.

```
student@bchd:~/mycode/flaskapi$ sudo apt install curl
```

32. Now that the server is running, try to curl against your API. The `-L` is required to follow the 3xx HTTP responses (forwarded).

```
student@bchd:~/mycode/flaskapi$ curl http://0.0.0.0:2224/user/admin -L
```

33. Did you follow what happened? If not, the following might help:

```
            http
curl                        flask
GET --------------------> /user/admin
<-----------------------  302 (go to /admin)
GET --------------------> /admin
<-----------------------  200 + plaintext
```

34. Try it again, only this time, supply the value `/Wolverine`

```
student@bchd:~/mycode/flaskapi$ curl http://0.0.0.0:2224/user/Wolverine -L
```

35. Consider what just happened:

```
            http
curl                        flask
GET --------------------> /user/wolverine
<-----------------------  302 (go to /guest/wolverine)
GET --------------------> /guest/wolverine
<-----------------------  200 + plaintext
```

Type `exit` to close your split screen window and return to your Flask app.

36.

37. Stop the Flask program with **ctrl** + `c`

38. Now let's try returning a separate HTML document. By default, documents are rendered from a sub directory called `templates`, so start by creating that directory.

```
student@bchd:~/mycode/flaskapi$ mkdir templates/
```

39. By default, the Flask route responds to the GET requests. However, this preference can be altered by providing methods argument to route() decorator. In order to demonstrate the use of POST method in URL routing, first let us create an HTML form and use the POST method to send form data to a URL. Create a new script called `postmaker.html`

```
student@bchd:~/mycode/flaskapi$ vim templates/postmaker.html
```

40. Create the following within `postmaker.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Sign in with your name</title>
    <form action = "/login" method = "POST">
        <p>Enter Name:</p>
        <p><input type = "text" name = "nm"></p>
        <p><input type = "submit" value = "submit"></p>
    </form>
</head>
<body>

</body>
</html>
```

41. Press **Esc** and then save and exit with `:wq`

42. Create a new script called `myflask04.py`

```
student@bchd:~/mycode/flaskapi$ vim myflask04.py
```

43. Create the following within `myflask04.py`

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   A simple Flask server. This server has the following endpoints:

   /success/<name> - responds with 200 + 'Welcome {name}'

   /
   /start          - Both endpoints respond with 200 + postmaker.html (template)

   /login          - a POST will have the form read for 'nm'
                   - a GET will be scanned for the query param ?nm=some_value"""

# python3 -m pip install flask
from flask import Flask
from flask import redirect
from flask import url_for
from flask import request
from flask import render_template

app = Flask(__name__)
## This is where we want to redirect users to
@app.route("/success/<name>")
def success(name):
    return f"Welcome {name}\n"
# This is a landing point for users (a start)
@app.route("/") # user can land at "/"
@app.route("/start") # or user can land at "/start"
def start():
    return render_template("postmaker.html") # look for templates/postmaker.html
# This is where postmaker.html POSTs data to
# A user could also browser (GET) to this location
@app.route("/login", methods = ["POST", "GET"])
def login():
    # POST would likely come from a user interacting with postmaker.html
    if request.method == "POST":
        if request.form.get("nm"): # if nm was assigned via the POST
            user = request.form.get("nm") # grab the value of nm from the POST
        else: # if a user sent a post without nm then assign value defaultuser
            user = "defaultuser"
    # GET would likely come from a user interacting with a browser
    elif request.method == "GET":
        if request.args.get("nm"): # if nm was assigned as a parameter=value
            user = request.args.get("nm") # pull nm from localhost:5060/login?nm=larry
        else: # if nm was not passed...
            user = "defaultuser" # ...then user is just defaultuser
    return redirect(url_for("success", name = user)) # pass back to /success with val for name
if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224) # runs the application
```

44. Press **Esc** and then save and exit with `:wq`

45. Run the script `myflask04.py`

```
student@bchd:~/mycode/flaskapi$ python3 myflask04.py
```

46. Split the screen by pressing **ctrl** + `b` and then **SHIFT** + `"`

47. Use `curl` against your API. The `-L` is required to follow the 3xx HTTP responses (forwarded). This will send a GET to `/login`

```
student@bchd:~/mycode/flaskapi$ curl http://0.0.0.0:2224/login?nm=Wolverine -L
```

48. You should receive back, `Welcome Wolverine`

49. Try it again, only this time, do not supply a value for `nm=`

```
student@bchd:~/mycode/flaskapi$ curl http://0.0.0.0:2224/login -L
```

50. You should receive back, `Welcome defaultuser`

51. Our service is also set up to accept a `POST`. To generate one using `curl`, include the `-d` flag, along with the `query=param` values you want to include in the POST form attachment.

```
student@bchd:~/mycode/flaskapi$ curl http://0.0.0.0:2224/login -L -d nm=Conan%20the%20Librarian
```

52. The application should respond back, `Hello Conan the Librarian`. However, if you look at the Flask service, you'll see a `POST` was sent, and *not* a `GET`.

53. Type `exit` to close the split screen session.

54. Stop your Flask app by pressing **ctrl** + `c`

55. That's it for this lab, good job.

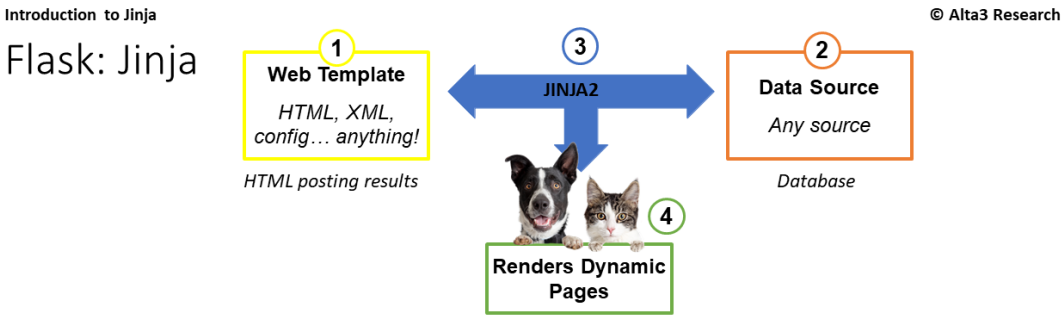
56. If you're tracking your code in a SCM, issue the following commands:
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "My first Flask apps"`
    - `git push origin`

34. Lecture - Introduction to Jinja

Lab Objective

The objective of this lab is to introduce the concept of Jinja templating.

Procedures



Jinja is a templating engine that is able to populate templates with data and use programming logic to make typically static documents (HTML, for example) render dynamically!

1. A flexible template is developed where its content is conditional on the data returned to it.
2. Data is read in, which can be anything from a simple variable to a database!
3. Jinja uses the logic in the template to pull the appropriate data from the source.
4. **Example**- A pet adoption agency wants to return a webpage with *N* animals, where *N* is defined by the number of adoptable animals within their database.

Moraa Onwonga
moraa.onwonga@accenturefederal.com

35. Flask APIs and Jinja2

Lab Objective

The objective of this lab is to continue to explore building APIs with Flask. In this example, we'll leverage the use of a Jinja2 template. Our API will first render an HTML template and return that in the HTTP request. This is an 'older' method of transmitting data. Today it's much more popular *just* to return JSON. The client can then parse out that JSON and display the data.

Jinja2 - A popular templating engine for Python. A web templating system combines a template with a certain data source to render dynamic web pages. The offical Jinja2 documentation can be found here: http://jinja.pocoo.org/docs/2.10/

Procedure

1. Move into (or create) a new directory to work in, `/home/student/mycode/flaskapi/`

   ```
   student@bchd:~$ mkdir -p /home/student/mycode/flaskapi/
   ```

2. Move into the new directory:

   ```
   student@bchd:~$ cd ~/mycode/flaskapi/
   ```

3. We can now take advantage of Jinja2 template engine on which Flask is based. Instead of returning hardcoded HTML from the function, a HTML file can be rendered by the `render_template()` function.

4. Now create a script called `jinja2temp01.py`

   ```
   student@bchd:~/mycode/flaskapi$ vim jinja2temp01.py
   ```

5. Copy the following into your Python script, `jinja2temp01.py`

   ```
   #!/usr/bin/env python3
   from flask import Flask, render_template
   app = Flask(__name__)

   @app.route("/")
   def index():
       return render_template("hellobasic.html")

   if __name__ == "__main__":
       app.run(host="0.0.0.0", port=2224)
   ```

6. Save and exit with `:wq`

7. Create a directory called `~/mycode/flaskapi/templates/`

   ```
   student@bchd:~/mycode/flaskapi$ mkdir ~/mycode/flaskapi/templates/
   ```

8. Move into the new directory. When you run your application from `~/mycode/flaskapi/` it will look for templates in the folder `~/mycode/flaskapi/template` by default.

   ```
   student@bchd:~/mycode/flaskapi$ cd ~/mycode/flaskapi/templates/
   ```

9. Now create a script called `hellobasic.html`

   ```
   student@bchd:~/mycode/templates/$ vim hellobasic.html
   ```

10. Copy and paste the following into your Jinja2 template, `hellobasic.html`

    ```
    <!doctype html>
    <html>
       <body>
          <h1>Hello!</h1>
       </body>
    </html>
    ```

11. Save and exit with `:wq`

12. Move back into your application directory.

    ```
    student@bchd:~/mycode/flaskapi/templates$ cd ~/mycode/flaskapi/
    ```

13. Run your script, `jinja2temp01.py`

    ```
    student@bchd:~/mycode/flaskapi$ python3 jinja2temp01.py
    ```

Split your screen with (CTRL + B) then (SHIFT + ")

14.

15. In the new screen cURL your API.

```
student@bchd:~$ curl http://0.0.0.0:2224 -L
```

16. Your screen should display the following:

```
<!doctype html>
<html>
   <body>
       <h1>Hello!</h1>
   </body>
</html>
```

17. Alternatively, you may visit `aux1` where the screen should display **Hello!** in bold.

18. Close the new window by typing `exit`.

```
student@bchd:~$ exit
```

19. Stop your Flask app by clicking on the terminal window it is running in and pressing `CTRL + C`

20. Now create a script called `jinja2temp02.py`

```
student@bchd:~/mycode/flaskapi$ vim jinja2temp02.py
```

21. Copy the following into your Python script, `jinja2temp02.py`

```
#!/usr/bin/env python3
from flask import Flask
from flask import render_template

app = Flask(__name__)

#grab the value 'username'
@app.route("/<username>")
def index(username):
    # render the jinja template "helloname.html"
    # apply the value of username for the var name
    return render_template("helloname.html", name = username)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=2224)
```

22. Save and exit with `:wq`

23. Now create a template called `templates/helloname.html`

```
student@bchd:~/mycode/flaskapi$ vim templates/helloname.html
```

24. Copy and paste the following into your Jinja2 template, `helloname.html`

```
<!doctype html>
<html>
   <body>
      <h1>Hello {{ name }}!</h1>
   </body>
</html>
```

25. Save and exit with `:wq`

26. Flask uses Jinja2 template engine. A web template contains HTML syntax interspersed with placeholders for variables and expressions (in these case Python expressions) which are replaced values when the template is rendered. In the above example, the variable `{{name}}` will be replaced. **Read about Jinja2 here:** http://jinja.pocoo.org/docs/2.10/ you may want to pay sepecial attention to the templates section: https://jinja.palletsprojects.com/en/2.10.x/templates/

27. Run your script, `jinja2temp02.py`

```
student@bchd:~/mycode/flaskapi$ python3 jinja2temp02.py
```

28. Split your screen with (CTRL + B) then (SHIFT + ")

29. In the new screen cURL your API. The `%20` is ASCII for whitespace.

```
student@bchd:~$ curl http://0.0.0.0:2224/James%20Bond -L
```

30. The screen should display the following:

```
<!doctype html>
<html>
   <body>
      <h1>Hello James Bond!</h1>
   </body>
</html>
```

31. Close the new window by typing `exit`.

```
student@bchd:~$ exit
```

32. Stop your Flask app by clicking on the terminal window it is running in and pressing `CTRL + C`

33. Now create a script called `jinja2temp03.py`

```
student@bchd:~/mycode/flaskapi$ vim jinja2temp03.py
```

34. Copy the following into your Python script, `jinja2temp03.py`

```
#!/usr/bin/python3
from flask import Flask
from flask import render_template

app = Flask(__name__)

# pull in the value of score as an int
@app.route("/scoretest/<int:score>")
def hello_name(score):
    # render the template with the value of score for marks
    # marks is a jinja var in the template
    return render_template("highscore.html", marks = score)

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224)
```

35. Save and exit with `:wq`

36. Now create a template called `templates/highscore.html`

```
student@bchd:~/mycode/flaskapi$ vim templates/highscore.html
```

37. Copy and paste the following into your Jinja2 template, `highscore.html`

```
<!doctype html>
<html>
   <body>

      {% if marks>50 %}
      <h1> You passed! Well done :)</h1>
      {% else %}
      <h1>You failed :(</h1>
      {% endif %}

   </body>
</html>
```

38. Save and exit with `:wq`

39. Note that the conditional statements if-else and endif are enclosed in delimiter `{%..%}`. **Read about Jinja2 conditionals here** http://jinja.pocoo.org/docs/2.10/templates/

40. Run your script, `jinja2temp03.py`

```
student@bchd:~/mycode/flaskapi$ python3 jinja2temp03.py
```

41. Split your screen with (CTRL + B) then (SHIFT + ")

42. In the new screen cURL your API.

```
student@bchd:~$ curl http://0.0.0.0:2224/scoretest/101 -L
```

43. The screen should display `<h1> You passed! Well done :)</h1>`

44. Now, try to cURL `http://0.0.0.0:2224/scoretest/42`.

```
student@bchd:~$ curl http://0.0.0.0:2224/scoretest/42 -L
```

45. `<h1>You failed :(</h1>` should be returned.

46. Finally, try to cURL `http://0.0.0.0:2224/scoretest/99`.

```
student@bchd:~$ curl http://0.0.0.0:2224/scoretest/99 -L
```

`<h1> You passed! Well done :)</h1>` should be returned.

47.

48. Close the new window by typing `exit`.

```
student@bchd:~$ exit
```

49. Stop your Flask app by clicking on the terminal window it is running in and pressing `CTRL + C`

50. HTML documents are not the ONLY thing we could render with Jinja. For many, it might be more practical to render something like a switch config. Create a new template that will contain a Cisco IOS switch configuration template.

```
student@bchd:~/mycode/flaskapi$ vim templates/baseIOS.conf.j2
```

51. Create the following Jinja template within `baseIOS.conf.j2`

```
!=== {{ switchname }} ===!

!--- IOS config ---!
enable
configure terminal
hostname {{ switchname }}

!--- MGMT ---!
username {{ username }} secret alta3
ip route 0.0.0.0 0.0.0.0 {{ defaultgateway }}
interface Management 1
ip address {{ switchIP }} {{ netmask }}
mtu {{ mtusize }}
exit

!--- SSH ---!
management ssh
  idle-timeout 0
  authentication mode keyboard-interactive
  server-port 22
  no fips restrictions
  no hostkey client strict-checking
  no shutdown
  login timeout 120
  log-level info
exit
exit
write memory
```

52. Save and exit with `:wq`

53. Let's write a new Flask application that can return a completed switch config.

```
student@bchd:~/mycode/flaskapi$ vim ciscoios.py
```

54. If you only need to support GET requests, no need to include the methods in your route decorator (such as POST). Everything beyond the "?" in your request is called a query parameter. Flask will take those query parameters out of the URL and place them into an ImmutableDict. You can access it with `request.args`, either with the key, `request.args["switchname"]` or with the get method. If you do use the get method, you expose an added ability to pass a default value (such as None), in the event it is not present. This is common for query parameters since they are often optional. Okay, now create the following Flask application:

```
#!/usr/bin/python3
from flask import Flask
from flask import render_template
from flask import request

app = Flask(__name__)

@app.route("/ciscoios/")
def ciscoios():
    try:
        qparms = {}
        # user passes switchname= or default "bootstrapped switch"
        qparms["switchname"]  = request.args.get("switchname", "bootstrapped switch")
        # user passes username= or default "admin"
        qparms["username"]  = request.args.get("username", "admin")
        # user passes gateway= or default "0.0.0.0"
        qparms["defaultgateway"] = request.args.get("gateway", "0.0.0.0")
        # user passes ip= or default "0.0.0.0"
        qparms["switchIP"] = request.args.get("ip", "0.0.0.0")
        # user passes mask= or default "255.255.255.0"
        qparms["netmask"] = request.args.get("mask", "255.255.255.0")
        # user passes mtu= or default "1450"
        qparms["mtusize"] = request.args.get("mtu", "1450")

        # render template and save as baseIOS.conf
        return render_template("baseIOS.conf.j2", **qparms)

    except Exception as err:
        return f"Uh-oh! {err}"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=2224)
```

55. Save and exit with `:wq`

56. Run your script, `ciscoios.py`

```
student@bchd:~/mycode/flaskapi$ python3 ciscoios.py
```

57. Split your screen with (CTRL + B) then (SHIFT + ")

58. In the new screen cURL your API.

```
student@bchd:~$ curl http://0.0.0.0:2224/ciscoios/ -L
```

59. The default Cisco IOS config should be returned.

60. Issue the following command. The command is long, so it wraps in the manual, however, the only space in the command is after `curl`. Customize yours to include query parameters with your lookup. Query parameters are separated with the ampersand, which also is a shell command, therefore, it is important that you quote the URI you construct.

```
 curl "http://0.0.0.0:2224/ciscoios/?switchname=hal9000&username=dreadpirateroberts&ip=192.168.0.1&mtu=1450&gateway=172.0.0.1" -L
```

61. Try reworking the curl command. Each time, check the file returned to see that it is being customized to your liking.

62. Close the new window by typing `exit`.

```
student@bchd:~$ exit
```

63. Stop your Flask app with (CTRL + C)

64. **Question: If you include erroneous query parameters in your lookup, do you still get back a CiscoIOS config?**

    - *Answer: YES! The parameters that are not needed are simply ignored*

65. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "Jinja templating for HTML and configs"`
    - `git push origin HEAD`

36. Jinja2 Challenge:

SCENARIO: You must maintain a hosts file that is constantly changing. Adding more hosts to the file should be easy to do (which means a lot of hard work on the back end!). Your goal is to use Flask to create output like this:

```
192.168.30.22 hostA.localdomain # hostA
192.168.30.33 hostB.localdomain # hostB
192.168.30.44 hostC.localdomain # hostB
```

Below is your "starter data" for this challenge. You'll put this inside your Flask API script as a normal list.

```
groups = [{"hostname": "hostA","ip": "192.168.30.22", "fqdn": "hostA.localdomain"},
          {"hostname": "hostB", "ip": "192.168.30.33", "fqdn": "hostB.localdomain"},
          {"hostname": "hostC", "ip": "192.168.30.44", "fqdn": "hostC.localdomain"}]
```

TASK 1:

- Create a Jinja2 template that would display the content of the `groups` value in the format shown at the top of this challenge.
- Create a Flask server that populates the Jinja2 template that you just made. Here is some starter code to get you going:

```
#!/usr/bin/python3

from flask import Flask
from flask import request
from flask import redirect
from flask import url_for
from flask import session
from flask import render_template

app = Flask(__name__)

app.secret_key= "random random RANDOM!"

groups = [{"hostname": "hostA","ip": "192.168.30.22", "fqdn": "hostA.localdomain"},
          {"hostname": "hostB", "ip": "192.168.30.33", "fqdn": "hostB.localdomain"},
          {"hostname": "hostC", "ip": "192.168.30.44", "fqdn": "hostC.localdomain"}]
```

TASK 2:

- Create a form that lets you add more data to `groups`- the data added to `groups` should be templatized as well!

TASK 3:

- Make an addition to your Flask server. Create a **session** that is made to include a **specific variable** (your choice- a password or user perhaps?). A user is ONLY able to add info to `groups` if that session variable is present.

37. Jinja2 Challenge SOLUTION:

The following is ONE possible solution to our Jinja2 challenge!

1. Create a directory for your template.

```
student@bchd:~$ mkdir -p ~/mycode/templates
```

2. Create your template:

```
student@bchd:~$ vim ~/mycode/templates/hosts.j2

{% for host in groups %}
   <h4><span>{{ host.ip }}   {{ host.fqdn }} # {{ host.hostname }}</span></h4>
   {% endfor %}

<br>
<li><a href = '/form'></b>Click here to add more data</b></a></li>
<li><a href = '/logout'></b>Click here to log out</b></a></li>
```

3. Create your form collector:

```
student@bchd:~$ vim ~/mycode/templates/formcollector.html.j2

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Enter new data</title>
    <form action = "/" method = "POST">
        <p>Enter Hostname:</p>
        <p><input type = "text" name = "hostname"></p>
        <p>Enter IP Address:</p>
        <p><input type = "text" name = "ip"></p>
        <p>Enter Fully Qualified Domain Name:</p>
        <p><input type = "text" name = "fqdn"></p>
        <p><input type = "submit" value = "submit"></p>
    </form>
</head>
<body>
</body>
</html>
```

4. Now create your Flask server:

```
student@bchd:~$ vim ~/mycode/solution.py
```

```
#!/usr/bin/python3

from flask import Flask
from flask import request
from flask import redirect
from flask import url_for
from flask import session
from flask import render_template

app = Flask(__name__)

app.secret_key= "random random RANDOM!"

groups = [{"hostname": "hostA","ip": "192.168.30.22", "fqdn": "hostA.localdomain"},
          {"hostname": "hostB", "ip": "192.168.30.33", "fqdn": "hostB.localdomain"},
          {"hostname": "hostC", "ip": "192.168.30.44", "fqdn": "hostC.localdomain"}]

@app.route("/", methods= ["GET","POST"])
def hosts():
    # GET returns the rendered hosts
    # POST adds new hosts, then returns rendered hosts
    if "username" in session and session["username"] == "admin":
        if request.method == "POST":
            # pull all values from posted form
            hostname = request.form.get("hostname")
            ip = request.form.get("ip")
            fqdn = request.form.get("fqdn")
            # create a new dictionary with values, add to groups
            groups.append({"hostname": hostname, "ip": ip, "fqdn": fqdn})
    return render_template("hosts.j2", groups=groups)

@app.route("/form", methods=["GET","POST"])
def form():
    # HTML form that collects hostname, ip, and fqdn values
    if request.method == "POST":
        session["username"] = request.form.get("username")
    if "username" in session and session["username"] == "admin":
        return render_template("formcollector.html.j2")
    else:
        return """
   <form action = "" method = "post">
      <p>Invalid Login.</p>
      <p><input type = text name = username></p>
      <p><input type = submit value = Login></p>
   </form>
  """

@app.route("/logout")
def logout():
    # accessing this page pops the value of username of the session
    session.pop("username", None)
    return redirect("/")

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224)
```

5. To test, start your Flask API.

```
student@bchd:~$ python3 ~/mycode/solution.py
```

6. Once Flask is running, go to your `aux1` page. Add new host values and confirm that they persist. Log in and out-- you must be `admin` to post changes!

38. Flask APIs and Cookies

Lab Objective

The objective of this lab is to explore Flask APIs and interacting with cookies. A cookie is stored on a client's computer in the form of a text file. Its purpose is to remember and track data pertaining to a client's usage for better visitor experience and site statistics.

A request object may contain a cookie's attribute(s). It is a dictionary object of all the cookie variables and their corresponding values. Additionally, the cookie will also usually store the expiry time, path, and domain name of the site it was issued from.

In Flask, cookies are set by a response object. Use the `make_response()` function to yield a a response object. After that, use the `set_cookie()` method to create the cookie.

Reading back a cookie is easy. The `get()` method of request.cookies attribute is used to read a cookie.

On the client side, cookies are typically managed by browsers. However, in this lab, we're going to learn to use `curl` to interact with the cookies returned by APIs. Before you begin, you should read about how `curl` manages cookies: https://curl.haxx.se/docs/http-cookies.html

Resources

- Python Flask - Cookies
- curl Documentation - HTTP and Cookies
- RFC 6265 - HTTP State Management Mechanism

Procedure

1. Move into (or create) a new directory to work in, `/home/student/mycode/flaskapi/`

   ```
   student@bchd:~$ mkdir -p /home/student/mycode/flaskapi/
   ```

2. Move into your new directory

   ```
   student@bchd:~$ cd ~/mycode/flaskapi/
   ```

3. Now create a script called `milkncookies.py`

   ```
   student@bchd:~/mycode/flaskapi$ vim milkncookies.py
   ```

4. Copy the following into your Python script, `milkncookies.py`

```
#!/usr/bin/env python3
"""Alta3 Research | rzfeeser@alta3.com
   Flask application that explores using cookies. This
   application has the following endpoints:

   /
   /login        - both endpoints return 200 + login.html (template)

   /setcookie    - POST returns 200 + cookie
                 - GET returns 302 redirect to /login

   /getcookie    - reads value of userID from client cookie
                   then returns 200 + '<h1>Welcome {name}</h1>'
                   where {name} is userID"""

# python3 -m pip install flask
from flask import Flask
from flask import make_response
from flask import request
from flask import render_template
from flask import redirect
from flask import url_for

app = Flask(__name__)

# entry point for our users
# renders a template that asks for their name
# login.html points to /setcookie
@app.route("/login")
@app.route("/")
def index():
    return render_template("login.html")

# set the cookie and send it back to the user
@app.route("/setcookie", methods = ["POST", "GET"])
def setcookie():
    # if user generates a POST to our API
    if request.method == "POST":
        if request.form.get("nm"): # if nm was assigned via the POST
        #if request.form["nm"] <-- this also works, but returns ERROR if no nm
            user = request.form.get("nm") # grab the value of nm from the POST
        else: # if a user sent a post without nm then assign value defaultuser
            user = "defaultuser"

        # Note that cookies are set on response objects.
        # Since you normally just return strings
        # Flask will convert them into response objects for you
        resp = make_response(render_template("readcookie.html"))
        # add a cookie to our response object
                        #cookievar #value
        resp.set_cookie("userID", user)

        # return our response object includes our cookie
        return resp

    if request.method == "GET": # if the user sends a GET
        return redirect(url_for("index")) # redirect to index

# check users cookie for their name
@app.route("/getcookie")
def getcookie():
    # attempt to read the value of userID from user cookie
    name = request.cookies.get("userID") # preferred method

    # name = request.cookies["userID"] # <-- this works but returns error
                                       # if value userID is not in cookie

    # return HTML embedded with name (value of userID read from cookie)
    return f'<h1>Welcome {name}</h1>'

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=2224)
```

5. Press **Esc** then save and exit with `:wq`

6. When you run your application, flask will look for templates within the folder, `templates/`, by default.

```
student@bchd:~/mycode/flaskapi/$ mkdir templates/
```

7. Within the templates folder, create a template called `login.html`.

```
student@bchd:~/mycode/flaskapi$ vim templates/login.html
```

8. Copy and paste the following into your template, `login.html`.

```
<!doctype html>
<html>
   <body>
      <form action = "/setcookie" method = "POST">
         <p><h3>Enter userID</h3></p>
         <p><input type = "text" name = "nm"/></p>
         <p><input type = "submit" value = 'Login'/></p>
      </form>
   </body>
</html>
```

9. Press **Esc** then save and exit with `:wq`

10. We are also going to need a template called `readcookie.html`. This will contain a simple redirection link to our getcookie resource.

```
student@bchd:~/mycode/flaskapi$ vim templates/readcookie.html
```

11. Copy and paste the following into your template, `readcookie.html`.

```
<!doctype html>
<html>
   <body>
      <h1> <p> A cookie has been set on your system! </p>
           <p> <a href="/getcookie">Click here to Read The Cookie</a> </p>
      </h1>
   </body>
</html>
```

12. Press **Esc** then save and exit with `:wq`

13. Run your script, `milkncookies.py`

```
student@bchd:~/mycode/flaskapi$ python3 milkncookies.py
```

14. Split your screen with **ctrl** + `b` then **Shift** + `"`

15. In the new screen cURL your API. The page asking for you name should be displayed.

```
student@bchd:~$ curl http://0.0.0.0:2224/ -L
```

16. Rework the cURL command to send a POST. This behavior mimics a user filling in their name `Larry`, and pressing the 'Submit' button.

```
student@bchd:~$ curl http://0.0.0.0:2224/setcookie -d "nm=Larry" -L
```

17. Looks like a cookie has been set on our system! Great. Let's try following that link to read the cookie on our system.

```
student@bchd:~$ curl http://0.0.0.0:2224/getcookie -L
```

18. Hmm, it looks like we're "Nobody". Time to dive a bit deeper into how cookies work.

19. The `curl` app has a built in "cookie engine" that mimics the Netscape navigator standard. In order to write a cookie to a file, we just use `-c` followed by the name of the cookie we want to create. The following command will accept the cookie information returned from the server, and write it into `cookie-jar.txt` (the name of the file is not important).

```
student@bchd:~$ curl http://0.0.0.0:2224/setcookie -d "nm=larry" -L -c cookie-jar.txt
```

20. Review `cookie-jar.txt`.

```
student@bchd:~$ cat cookie-jar.txt
```

21. Looks like information was recorded on our system (in our cookie) that we are larry. Notice the IP address of the server was also provided (in production, we'd likely see a domain).

22. Once again, let's try access `/getcookie`, only this time, we'll make our cookie, `cookie-jar.txt` available to Flask.

```
student@bchd:~$ curl http://0.0.0.0:2224/getcookie -L -b cookie-jar.txt
```

23. And there you go! The system should now recognize you as Larry! If you had used a browser, the browser would of managed the cookie, and made it available to the API (website) you were visiting.

24. Try changing the value found in your cookie. You can make `nm=` to anything you would like.

```
student@bchd:~$ curl http://0.0.0.0:2224/setcookie -d "nm=Frodo%20Baggins" -L -c cookie-jar.txt
```

25. Review `cookie-jar.txt`. Notice it has been updated.

```
student@bchd:~$ cat cookie-jar.txt
```

26. Try to access `/getcookie`. Make the cookie, `cookie-jar.txt`, available to Flask.

```
student@bchd:~$ curl http://0.0.0.0:2224/getcookie -L -b cookie-jar.txt
```

The system should send a greeting to your new user.

27.
28. **CHALLENGE**: Manually hack your cookie, and then have the new user returned by the API `/getcookie`. *HINT: Within cookie-jar.txt, replace the value represented by* `userID` *to something else. When you finish, try to cURL* `/getcookie`*. Be sure to include your cookie!*

29. Close the new window by typing `exit`

    ```
    student@bchd:~$ exit
    ```

30. Stop your Flask app with **ctrl** + `c`

31. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "serving cookies"`
    - `git push origin`

39. Flask Sessions

Lab Objective

The objective of this lab is to explore Flask's ability to leverage sessions. A Flask Session is yet another way to store user-specific data between requests. This is similar to cookies, but with a different intended case use.

Cookies are intended to be long-term. Case use could be tracking a users location, or perhaps a preference for JSON versus CSV being returned from an API.

Flask sessions (which are implemented as cookies) are intended for temporary storage of session data. Generally, the session data includes a way to invalidate itself once the browser closes, or after a short period of time (say a half hour). A good use of session cookies could be items in a user's cart.

To create a session, you must set a secret key within Flask. While Flask does perform a basic encryption of session data, it is not secure (YouTube has lots of videos of people decoding session cookies). Once created, the session object of the flask package is used to set and get session data. The session object works like a dictionary but it can also keep track of modifications.

For example, to set a 'username' session variable use the statement `session["username"] = "admin"`. To release a session variable use the `pop()` method, as in, `session.pop("username", None)`.

Resources

- Flask Documentation - Sessions
- GitHub.com/RZFeeser - API Quest - A simple RPG created by the course author using Flask. The project uses sessions to maintain state as a user plays through the game.

Procedure

1. Move into (or create) a new directory to work in, `/home/student/mycode/flaskapi/`

   ```
   student@bchd:~$ mkdir ~/mycode/flaskapi/
   ```

2. Move into the new directory.

   ```
   student@bchd:~$ cd ~/mycode/flaskapi/
   ```

3. Now create a script called `session01.py`

   ```
   student@bchd:~/mycode/flaskapi$ vim session01.py
   ```

4. The following code is a simple demonstration of session works in Flask. URL `/` simply prompts user to log in, as session variable "username" is not set. Copy the following into your Python script, `session01.py`:

```
#!/usr/bin/python3

from flask import Flask
from flask import session
from flask import render_template
from flask import redirect
from flask import url_for
from flask import escape
from flask import request

app = Flask(__name__)
app.secret_key = "any random string"

## If the user hits the root of our API
@app.route("/")
def index():
  ## if the key "username" has a value in session
  if "username" in session:
    username = session["username"]
    return "Logged in as " + username + "<br>" + \
      "<b><a href = '/logout'>click here to log out</a></b>"

  ## if the key "username" does not have a value in session
  return "You are not logged in <br><a href = '/login'></b>" + \
      "click here to log in</b></a>"

## If the user hits /login with a GET or POST
@app.route("/login", methods = ["GET", "POST"])
def login():
   ## if you sent us a POST because you clicked the login button
   if request.method == "POST":

      ## request.form["xyzkey"]: use indexing if you know the key exists
      ## request.form.get("xyzkey"): use get if the key might not exist
      session["username"] = request.form.get("username")
      return redirect(url_for("index"))

   ## return this HTML data if you send us a GET
   return """
   <form action = "" method = "post">
      <p><input type = text name = username></p>
      <p><input type = submit value = Login></p>
   </form>
  """

@app.route("/logout")
def logout():
   # remove the username from the session if it is there
   session.pop("username", None)
   return redirect(url_for("index"))

if __name__ == "__main__":
  app.run(host="0.0.0.0", port=2224)
```

5. Save and exit with `:wq`

6. If a user browses to the root, `/`, they will be notified they are not logged in, and be prompted to click on a link directing them to, `"/login"`. Upon this click, the user is returned HTML containing a web form. When the returned form is submitted, it is sent to `"/login"` with a POST. This causes a session variable to be set. The application is then redirected to `/`, however, this time session variable `username` is found.

7. The application also contains a logout() view function, which pops out `username` session variable. `/` URL again shows the opening page.

8. Run your script, `session01.py`

```
student@bchd:~/mycode/flaskapi$ python3 session01.py
```

9. Split your screen with (CTRL + B) then (SHIFT + ")

10. Start by sending a cURL to the root. Notice that we get back HTML that would ask us to click a link to go to `/login`

```
student@bchd:~$ curl http://0.0.0.0:2224/ -L
```

11. Mimic clicking the link to follow `/login`

```
student@bchd:~$ curl http://0.0.0.0:2224/login -L
```

12. Notice we were returned a web form. Mimic filling out this form, and submitting it via a POST. We also want to accept our session cookie and write the data in `session-cookie.txt`.

```
student@bchd:~$ curl http://0.0.0.0:2224/login -L -d "username=Homer" -c session-cookie.txt
```

13. Try viewing `session-cookie.txt` the data should have basic encryption that prevents effortless tampering.

```
student@bchd:~$ cat session-cookie.txt
```

14. Use cURL to access the root again. You should be logged in as the name you assigned yourself within this session.

```
student@bchd:~$ curl http://0.0.0.0:2224/ -L -b session-cookie.txt
```

15. Unfortunately, a small limitation of cURL is that our logout function will not work. However, it *does* work with browsers! Ask your instructor if you'd like to see a demo of the functionality.

16. Exit your second window.

```
student@bchd:~$ exit
```

17. Stop your Flask app by clicking on the terminal window it is running in and pressing `CTRL + C`

18. **Helpful Hint** The secret key (random string) within your script should be as random as possible. Your operating system can generate random data if you run the following command:

```
student@bchd:~/mycode/flaskapi$ python3 -c "import os; print(os.urandom(16))"
```

19. Remember, if the random string changes between runs, then all previous session cookies will be invalidated!

20. **CHALLENGE 01** - Tweak the session so it stores the name and age of a person. Don't forget that you'll need to update your input mechanism to include age.

21. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "Flask sessions"`
    - `git push origin`

40. LECTURE - Controlling your APIs

Lecture Objective

The objective of this lecture is to learn to apply control to your APIs, including limiting, selecting custom failure codes, as well as handling redirection. Incorporating this content will have us explore:

- **redirect() function:** when called, it returns a response object and redirects the user to another target location with specified status code.
    - The full syntax would be `Flask.redirect(location, statuscode, response)`. The `location` parameter is the URL where response should be redirected. The `statuscode` is sent to the browser's header, which defaults to 302. The `response` parameter is used to instantiate response.

> The following status codes are standardized: *HTTP_300_MULTIPLE_CHOICES, HTTP_301_MOVED_PERMANENTLY, HTTP_302_FOUND, HTTP_303_SEE_OTHER, HTTP_304_NOT_MODIFIED, HTTP_305_USE_PROXY, HTTP_306_RESERVED, HTTP_307_TEMPORARY_REDIRECT*

- **abort() function:** when called, an error will be returned to the client. As an argument, you get to specify which 4xx status code should be received. Here are a few you may wish to incorporate when appropriate:

```
400 for Bad Request

401 for Unauthenticated

403 for Forbidden

404 for Not Found

406 for Not Acceptable

415 for Unsupported Media Type

429 for Too Many Requests
```

- Limiting is a way to throttle our users from over-consuming data. This may be unintentional, or malicious in nature (think DOS attacking). Flask API limiting is best handled by the following package.
- **File upload in Flask** is very easy! This requires an HTML form with its enctype attribute set to `multipart/form-data` and posts the file to a URL. The URL handler fetches file from `request.files[]` object and saves it to the desired location.
    - Each uploaded file is first saved in a temporary location on the server before it is actually saved to its ultimate location. The name of destination file can be hard-coded or can be obtained from filename property of `request.files[file]` object. However, it is recommended to obtain a secure version of it using the `secure_filename()` function.
    - It is possible to define the path of default upload folder and maximum size of uploaded file in configuration settings of Flask object. The code `app.config['UPLOAD_FOLDER']` will define the path for upload folder, whereas `app.config['MAX_CONTENT_PATH']` specifies the maximum size of file to be uploaded in bytes.
    - Read more about uploading files with Flask: https://flask.palletsprojects.com/en/1.1.x/patterns/fileuploads/

Demonstration

1. We'll start by ensuring we have a `templates` directory to work from in our current working directory.

```
student@bchd:~$ mkdir -p ~/demo/flaskapi/templates
```

2. The following is an HTML form asking a VERY IMPORTANT QUESTION that some of you may recognize!

```
student@bchd:~$ vim ~/demo/flaskapi/templates/towel.html
```

3. Copy and paste the following into the template:

```
<html>
<style>
body {
  background-color: black;
  text-align: center;
  color: white;
  font-family: Arial, Helvetica, sans-serif;
}
</style>
</head>
<body>

<h1>TRIVIA TIME</h1>
<p>What is the answer to life, the universe, and everything?</p>
<img src={{ pic }} alt="Avatar" style="width:200px">
    <form action = "/login" method = "POST">
        <p><input type = "text" name = "answer"></p>
        <p><input type = "submit" value = "submit"></p>
    </form>

</body>
</html>
```

4. We'll now create a script called `~/demo/flaskapi/redirectdemo.py`

```
student@bchd:~$ vim ~/demo/flaskapi/redirectdemo.py
```

5. In the following example, the `redirect()` function will send us to an error if the wrong answer is provided.

```
#!/usr/bin/python3
"""Alta3 Research
Simple flask application using redirect()"""

from flask import Flask
from flask import redirect
from flask import url_for
from flask import render_template
from flask import request
from flask import abort

app = Flask(__name__)

pic_location= "https://static.alta3.com/courses/api/lec_flaskcontrol_python/dont-panic.png"

# if user sends GET to / (root)
@app.route("/")
def index():
    return render_template("towel.html", pic= pic_location)   # found in templates/

# if user sends GET or POST to /login
@app.route("/login", methods = ["POST", "GET"])
def login():
    # if user sent a POST
    if request.method == "POST":
        # if the POST contains '42' as the value for 'answer'
        if request.form["answer"] == "42" :
            return redirect(url_for("success")) # return a 302 redirect to /success
        else:
            return redirect(url_for("fail"))    # return a 302 redirect to /fail
    elif request.method == "GET":
        return redirect(url_for("index")) # if they sent a GET to /login send 302 redirect to /

@app.route("/httpfail")
def httpfail():
    abort(406)  # send back a HTTP failure

@app.route("/fail")
def fail():
    return "That was not correct." # nothing wrong with HTTP layer, we just indicating that the user responded incorrectly

@app.route("/success")
def success():
    return "Correct!"

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224)
```

6. Save and exit with `:wq`

7. Run your script, `~/demo/flaskapi/redirectdemo.py`

```
student@bchd:~$ python3 ~/demo/flaskapi/redirectdemo.py
```

8. Split your screen with (CTRL + B) then (SHIFT + ") *alternatively, use a TTY session*

9. In the new screen cURL your API with a GET. The root should return a webform, which would be of interest to users accessing your API with a browser. Notice that use of this form will result in sending a POST to `/login`

```
student@bchd:~$ curl http://0.0.0.0:2224 -L
```

10. In the new screen cURL your API by generating a POST. This step mimics using a browser to click the "Submit" button. This should NOT work, as `coffee` is not the correct answer. You will get a 200 response, but the attached text will indicate the answer is wrong.

```
student@bchd:~$ curl http://0.0.0.0:2224/login -d "answer=coffee" -L
```

11. In the new screen cURL your API by generating a POST. This should will return text indicating you were correct, as 42 is, of course, the answer!

```
student@bchd:~$ curl http://0.0.0.0:2224/login -d "answer=42" -L
```

12. API limiting is a way to throttle how fast a user can consume our API resources. There is good reason to do this, after all, lockups aren't free! Flask API limiting is best handled by Flask limiter. First, install Flask Limiter.

```
student@bchd:~$ python3 -m pip install Flask-Limiter
```

13. We'll now make a change to our previous script... let's only give the user 3 guesses before they are cut off from making additional requests!

```
student@bchd:~$ vim ~/demo/flaskapi/redirectdemo.py
```

```
#!/usr/bin/python3
"""Alta3 Research
Example of applying limiting to our flask APIs using Flask-Limiter"""

# NEW
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

from flask import Flask
from flask import redirect
from flask import url_for
from flask import render_template
from flask import request
from flask import abort

app = Flask(__name__)

pic_location= "https://static.alta3.com/courses/api/lec_flaskcontrol_python/dont-panic.png"

# create a limiter object from Limiter
# limits are being performed by tracking the
# REMOTE ADDRESS of the clients
limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["200 per day", "50 per hour"]
)

# if user sends GET to / (root)
@app.route("/")
def index():
    return render_template("towel.html", pic= pic_location)   # found in templates/

# if user sends GET or POST to /login
@app.route("/login", methods = ["POST", "GET"])
@limiter.limit("3 per day")
def login():
    # if user sent a POST
    if request.method == "POST":
        # if the POST contains '42' as the value for 'answer'
        if request.form["answer"] == "42" :
            return redirect(url_for("success")) # return a 302 redirect to /success
        else:
            return redirect(url_for("fail"))    # return a 302 redirect to /fail
    elif request.method == "GET":
        return redirect(url_for("index")) # if they sent a GET to /login send 302 redirect to /

@app.route("/httpfail")
def httpfail():
    abort(406)  # send back a HTTP failure

@app.route("/fail")
def fail():
    return "That was not correct." # nothing wrong with HTTP layer, we just indicating that the user responded incorrectly

@app.route("/success")
def success():
    return "Correct!"

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224)
```

14. Now let's try our changes to the server.

```
student@bchd:~$ python3 ~/demo/flaskapi/redirectdemo.py
```

15. Split your screen with (CTRL + B) then (SHIFT + ")

16. In the new screen cURL your API with three of our favorite things as WRONG answers.

```
student@bchd:~$ curl http://0.0.0.0:2224/login -d "answer=Raindrops on roses" -L

student@bchd:~$ curl http://0.0.0.0:2224/login -d "answer=Whiskers on kittens" -L

student@bchd:~$ curl http://0.0.0.0:2224/login -d "answer=Bright copper kettles" -L

student@bchd:~$ curl http://0.0.0.0:2224/login -d "answer=Warm woolen mittens" -L
```

> Looks like we've hit the limit! As you can see, limitation can be described as either a default value or customized per path. You'll have the chance to explore this on your own in the next lab!

17. For the last part of this demonstration, we'll explore how one can upload files to a Flask server. To select the file, we'll create an HTML form to do this.

```
student@bchd:~/mycode/flaskapi$ vim ~/demo/flaskapi/templates/upload.html
```

```
<html>
   <body>
      <form action = "/uploader" method = "POST"
         enctype = "multipart/form-data">
         <input type = "file" name = "file" />
         <input type = "submit"/>
      </form>
   </body>
</html>
```

18. With all of these files, let's stay organized. Create a `static` directory to hold our files.

```
student@bchd:~$ cd ~/demo/flaskapi && mkdir static
```

19. Let's go back and tweak our script again. What we'll do is add the `/upload` URL rule to display our `upload.html` we just made. We'll also add the `/upload-file` URL rule that, when called, will allow us to upload a picture from our personal machine and set it as the new background for our question page!

```
student@bchd:~/mycode/flaskapi$ vim ~/demo/flaskapi/redirectdemo.py
```

20. Create the following:

```
#!/usr/bin/python3
"""Alta3 Research
Adding upload functionality to our scripting"""

from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
from flask import Flask
from flask import redirect
from flask import url_for
from flask import render_template
from flask import request
from flask import abort

app = Flask(__name__)

pic_location= "https://static.alta3.com/courses/api/lec_flaskcontrol_python/dont-panic.png"

limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["200 per day", "50 per hour"]
)

@app.route("/upload")
def upload():
  print(pic_location)
  return render_template("upload.html")

@app.route("/uploader", methods = ["GET","POST"])
def upload_file():
  global pic_location
  if request.method == "GET":  # if method is a get (same as "/upload")
     return render_template("upload.html")
  if request.method == "POST":
     f = request.files["file"]
     filename= f.filename
     file_ext= filename.split(".")[-1]
     pic_location= f"static/newpic.{file_ext}"
     f.save("static/newpic." + file_ext)
     return redirect("/")

# if user sends GET to / (root)
@app.route("/")
def index():
    return render_template("towel.html", pic= pic_location)   # found in templates/

# if user sends GET or POST to /login
@app.route("/login", methods = ["POST", "GET"])
@limiter.limit("3 per day")
def login():
    # if user sent a POST
    if request.method == "POST":
        # if the POST contains '42' as the value for 'answer'
        if request.form["answer"] == "42" :
            return redirect(url_for("success")) # return a 302 redirect to /success
        else:
            return redirect(url_for("fail"))    # return a 302 redirect to /fail
    elif request.method == "GET":
        return redirect(url_for("index")) # if they sent a GET to /login send 302 redirect to /

@app.route("/httpfail")
def httpfail():
    abort(406)  # send back a HTTP failure

@app.route("/fail")
def fail():
    return "That was not correct." # nothing wrong with HTTP layer, we just indicating that the user responded incorrectly

@app.route("/success")
def success():
    return "Correct!"

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224)
```

21. Let's try this out!

```
student@bchd:~/mycode/flaskapi$ python3 ~/demo/flaskapi/redirectdemo.py
```

22. In your `Group Dashboard` tab, open `aux1` in a separate window. Add `/upload` to the end of the URL at the top of your browser.

23. From your personal machine, select an image to upload! Click `Submit` after choosing file. Form's POST method invokes the `/uploader` URL. The underlying function `uploader()` does the save operation. The new file will be rendered in the template when redirected back to our question page!

24. Hopefully this was a fun introduction into using these tools. In the following lab, you'll have the opportunity to try them out yourself!

41. Flask Redirection, Errors, and API Limiting

Lab Objective

The objective of this lab is to explore redirection, errors and API limiting. Flask class has a `redirect()` function. When called, it returns a response object and redirects the user to another target location with specified status code.

Consider `Flask.redirect(location, statuscode, response)`. The `location` parameter is the URL where response should be redirected. The `statuscode` is sent to the browser's header, which defaults to 302. The `response` parameter is used to instantiate response.

The following status codes are standardized: *HTTP_300_MULTIPLE_CHOICES, HTTP_301_MOVED_PERMANENTLY, HTTP_302_FOUND, HTTP_303_SEE_OTHER, HTTP_304_NOT_MODIFIED, HTTP_305_USE_PROXY, HTTP_306_RESERVED, HTTP_307_TEMPORARY_REDIRECT*

Limiting is a way to throttle our users from over-consuming data. This may be unintentional, or malicious in nature (think DOS attacking). Flask API limiting is best handled by the Flask plugin package Flask Limiter.

Resources

- Flask plugin - Flask Limiter
- RFC 7231 HTTP Semantics and Content - Status Codes

Procedure

1. Start off in your home directory

```
student@bchd:~$ cd
```

2. Create a new directory to work in, `/home/student/mycode/flaskapi/`

```
student@bchd:~$ mkdir -p /home/student/mycode/flaskapi/
```

3. You'll also need a templates directory for our jinja template.

```
student@bchd:~$ mkdir -p /home/student/mycode/flaskapi/templates/
```

4. Now create the template.

```
student@bchd:~$ vim ~/mycode/flaskapi/templates/log_in.html
```

5. Copy and paste the following into the template:

```
<html>
        <body>
            <form action = "/login" method = "POST">
            <p><h3>Enter your login credential</h3></p>
            <p><input type = "text" name = "username"/></p>
            <p><input type = "submit" value = "Login"/></p>
            </form>
        </body>
</html>
```

6. Press **Esc** and then save and exit with `:wq`

7. Move into the new directory.

```
student@bchd:~$ cd ~/mycode/flaskapi/
```

8. Now create a script called `redirect01.py`

```
student@bchd:~/mycode/flaskapi$ vim redirect01.py
```

9. In the following example, the `redirect()` function is used to display the login page again when a login attempt fails. The failure will occur if anyone except admin tries to log in. Copy the following into your Python script, `redirect01.py`:

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com

Making use of HTTP non-200 type responses.
https://tools.ietf.org/html/rfc2616 # rfc spec describing HTTP
1xx - informational
2xx - success / ok
3xx - redirection
4xx - errors
5xx - server errors
"""

from flask import Flask
from flask import redirect
from flask import url_for
from flask import render_template
from flask import request
from flask import abort

app = Flask(__name__)

# if user sends GET to / (root)
@app.route("/")
def index():
    return render_template("log_in.html")   # found in templates/

# if user sends GET or POST to /login
@app.route("/login", methods = ["POST", "GET"])
def login():
    # if user sent a POST
    if request.method == "POST":
        # if the POST contains 'admin' as the value for 'username'
        if request.form["username"] == "admin" :
            return redirect(url_for("success")) # return a 302 redirect to /success
        else:
            abort(401)  # if they didn't supply the username 'admin' send back a 401, auth failure
    elif request.method == "GET":
        return redirect(url_for("index")) # if they sent a GET to /login send 302 redirect to /

@app.route("/success")
def success():
    return "logged in successfully"

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224)
```

10. Press **Esc** and then save and exit with `:wq`

11. Run your script, `redirect01.py`

```
student@bchd:~/mycode/flaskapi$ python3 redirect01.py
```

12. Split your screen with **ctrl** + `b` and then **shift** + `"`

13. In the new screen cURL your API with a GET. The root should return a webform, which would be of interest to users accessing your API with a browser. Notice that use of this form will result in sending a POST to `/login`

```
student@bchd:~$ curl http://0.0.0.0:2224 -L
```

14. In the new screen cURL your API by generating a POST. This step mimics using a browser to click the "Submit" button. This should NOT work, as `cupcake` is not a valid user. You will get a 401 response.

```
student@bchd:~$ curl http://0.0.0.0:2224/login -d "username=cupcake" -L
```

15. In the new screen cURL your API by generating a POST. This should SHOULD work, as admin is a valid user.

```
student@bchd:~$ curl http://0.0.0.0:2224/login -d "username=admin" -L
```

16. Close the new window by typing `exit`.

```
student@bchd:~$ exit
```

17. Stop your Flask app by clicking on the terminal window it is running in and pressing **ctrl** + `c`

18. If you want to try out some other error codes, try the following in place of the 401:

```
400 for Bad Request

401 for Unauthenticated

403 for Forbidden

404 for Not Found

406 for Not Acceptable

415 for Unsupported Media Type

429 for Too Many Requests
```

19. API limiting is a way to throttle how fast a user can consume our API resources. There is good reason to do this, after all, lockups aren't free! Flask API limiting is best handled by Flask limiter. First, install Flask Limiter.

```
student@bchd:~/mycode/flaskapi$ python3 -m pip install Flask-Limiter
```

20. Create a new script:

```
student@bchd:~/mycode/flaskapi$ vim limitedapis.py
```

21. Create the following script. Be sure to read the comments:

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   Using the Flask-Limiter package to set limits
   on individual API requests from an IP."""

from flask import Flask
## from python3 -m pip install Flask-Limiter
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

# create an app object from Flask
app = Flask(__name__)

# create a limiter object from Limiter
# limits are being performed by tracking the
# REMOTE ADDRESS of the clients
limiter = Limiter(
    app = app,
    key_func=get_remote_address,
    default_limits=["200 per day", "50 per hour"]
)

# we now have TWO decorators on our function
# app.route() describes WHEN to trigger the function
# limiter.limit() describes HOW OFTEN to trigger the function
@app.route("/slow")
@limiter.limit("1 per day")
def slow():
    return "Enjoy this message. It will only display once per day."

# No limiter decorator is needed, this function STILL is limited
# by 200 lookups per day, and 50 per hour
@app.route("/fast")
def fast():
    return "I inherit the default limits of 200 per day and 50 per hour."

## limiter().exempt removes all limits on this API
@app.route("/ping")
@limiter.exempt
def ping():
    return "PONG FOREVER!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=2224)
```

22. Press **Esc** and then save and exit with `:wq`

23. Try running your server.

```
student@bchd:~/mycode/flaskapi$ python3 limitedapis.py
```

24. Split your screen with (CTRL + B) then (SHIFT + ")

25. In the new screen cURL your API with a GET to /slow. This will work once.

```
student@bchd:~$ curl http://0.0.0.0:2224/slow
```

26. Try it again. This time the limiter will cause a fail.

```
student@bchd:~$ curl http://0.0.0.0:2224/slow
```

27. Try out the other endpoints, `/fast`. This will work 200 times per day, 50 times per hour.

```
student@bchd:~$ curl http://0.0.0.0:2224/fast
```

28. The other endpoint was `/ping`. This is excluded from our limiting. Therefore, `/ping` works forever.

```
student@bchd:~$ curl http://0.0.0.0:2224/ping
```

29. Close the new window by typing `exit`.

```
student@bchd:~$ exit
```

30. Stop your Flask app by clicking on the terminal window it is running in and pressing `CTRL + C`

31. **CHALLENGE 01 (OPTIONAL)** - Write a client that can issue 51 lookups to `/fast`. The last lookup should fail.

32. *SOLUTION 01 - The following is a possible solution to CHALLENGE 01.*

```
#!/usr/bin/python3
"""Alta3 Research | RZFeeser
   SOLUTION 01 - How quickly can we get to 50 requests (per hour) to be rate limited
   by FlaskLimiter? Run the script to get results."""

import time
import requests

URI = "http://localhost:2224/"

def main():

    # get the current time
    start_time = time.time()

    # start an infinite loop
    while True:
        r = requests.get(f"{URI}fast")  # this URI is limited by 50 lookups per hour
        if r.status_code != 200:
            end_time = time.time()
            break # stop looping, as we have hit the limit

    # display the total time it took to perform the lookups
    print(f"To reach the limit of /fast, it took {end_time - start_time} seconds")

# invoke the main function
if __name__ == "__main__":
    main()
```

33. That's it for this lab.

34. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "Errors redirects and limits"`
    - `git push origin`

42. Flask Uploading and Downloading Files

Lab Objective

The objective of this lab is to explore how to upload files to an API with Flask. Handling file upload in Flask is very easy. It needs an HTML form with its enc-type attribute set to `multipart/form-data`, posting the file to a URL. The URL handler fetches file from `request.files[]` object and saves it to the desired location.

Each uploaded file is first saved in a temporary location on the server before it is actually saved to its ultimate location. The name of destination file can be hard-coded or can be obtained from filename property of `request.files[file]` object. However, it is recommended to obtain a secure version of it using the `secure_filename()` function.

It is possible to define the path of default upload folder and maximum size of uploaded file in configuration settings of Flask object. The code `app.config['UPLOAD_FOLDER']` will define the path for upload folder, whereas `app.config['MAX_CONTENT_PATH']` specifies the maximum size of file to be uploaded in bytes.

Sometimes, we also want download data to be made highly available, such a graph showing server information. In this lab, we'll create a second application able to SSH into various servers, retrieve their up time, and then graph the results. This cycle will be repeated every 30 seconds, and made highly accessible via our Flask HTTP framework.

Resources

- Python Flask - Uploading Files
- Python Matplotlib - Visual Graphing Homepage

Procedure

1. Start off in the home directory

```
student@bchd:~$ cd
```

2. Create a new directory to work in, `/home/student/mycode/flaskapi/`

```
student@bchd:~$ mkdir -p ~/mycode/flaskapi/templates
```

3. Move into a new directory to work in, `/home/student/mycode/flaskapi/`

```
student@bchd:~$ cd ~/mycode/flaskapi/
```

4. Now create a script called `uploader01.py`

```
student@bchd:~/mycode/flaskapi$ vim uploader01.py
```

5. The following code has the `/upload` URL rule that displays `upload.html` from the templates folder. It also contains the `/upload-file` URL rule that calls `upload_file()` function for handling the upload process. Copy the following into your Python script, `uploader01.py`:

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   A simple Flask server to upload and download files"""

# python3 -m pip install flask
from flask import Flask
from flask import render_template
from flask import request
from werkzeug.utils import secure_filename

app = Flask(__name__)

# Return HTML that allows file uploads
@app.route("/upload")
def upload():
  return render_template("upload.html")

# this endpoint accepts the upload (POST) or download (GET)
@app.route("/uploader", methods = ["GET", "POST"])
def upload_file():
  if request.method == "GET":  # if method is a get (same as "/upload")
     return render_template("upload.html")
  if request.method == "POST":
     f = request.files["file"]
     f.save(secure_filename(f.filename))
     return "file uploaded successfully"

if __name__ == "__main__":
   app.run(host="0.0.0.0", port=2224)
```

6. Press **Esc** and then save and exit with `:wq`

7. We need a template, `upload.html`. Create this now.

```
student@bchd:~/mycode/flaskapi$ vim templates/upload.html
```

8. Copy and paste the following into your script.

```
<html>
   <body>
      <form action = "/uploader" method = "POST"
         enctype = "multipart/form-data">
         <input type = "file" name = "file" />
         <input type = "submit"/>
      </form>
   </body>
</html>
```

9. Press **Esc** and then save and exit with `:wq`

10. Run your script, `uploader01.py`

```
student@bchd:~/mycode/flaskapi$ python3 uploader01.py
```

11. In your `Group Dashboard` tab, open `aux1` in a separate window. Add `/upload` to the end of the URL at the top of your browser.

12. If you like, select a file to upload! Click `Submit` after choosing file. Form's POST method invokes the `/uploader` URL. The underlying function `uploader()` does the save operation.

13. The file you just uploaded is now in the same directory you ran your script from: `~/mycode/flaskapi/`

14. Stop your Flask app by clicking on the terminal window it is running in and pressing **ctrl** + `c`

15. Look for the file you uploaded in your local directory.

```
student@bchd:~/mycode/flaskapi$ ls
```

16. Suppose you wanted to make a file that was continually returned, like a graph. This graph needs to be returned every 30 seconds and display the up times of some servers.

17. To start, make sure you created a `static/` directory to save and retrieve files in, similar to your `templates/` directory. Flask allows us to return files, like images, from this directory.

```
student@bchd:~/mycode/flaskapi$ mkdir /home/student/mycode/flaskapi/static/
```

18. Install our various Python3 dependencies. First, numpy (say it, "num-pie").

```
student@bchd:~/mycode/flaskapi$ python3 -m pip install numpy
```

19. Make sure pyyaml is installed.

```
student@bchd:~/mycode/flaskapi$ python3 -m pip install pyyaml
```

20. Make sure paramiko is installed.

```
student@bchd:~/mycode/flaskapi$ python3 -m pip install paramiko
```

21. Make sure matplotlib is installed. Matplotlib allows us to quickly create excel quality graphs and charts as PDFs and PNGs.

```
student@bchd:~/mycode/flaskapi$ python3 -m pip install matplotlib
```

22. Install Ansible. This is an automation framework written in Python. We'll use it to deploy our targets.

```
student@bchd:~/mycode/flaskapi$ python3 -m pip install ansible
```

23. To recap, our application will SSH into some servers, retrieve their up time, and then graph the results. This cycle will be repeated every 30 seconds, and accessible by HTTP. For this to work, we'll need some Linux servers to log into. Move into your home directory.

```
student@bchd:~/mycode/flaskapi$ cd ~/
```

24. Run a script that starts new SSH endpoints, "Bender", "Fry", "Zoidberg", and "Farnsworth":

```
student@bchd:~$ bash ~/px/scripts/full-setup.sh
```

25. Now that we have SSH targets, move back into our working directory.

```
student@bchd:~$ cd /home/student/mycode/flaskapi/
```

26. We need to save our credentials to log into those servers in a secure place. Let's store them as a yaml format in, `/home/student/sshpass.yml`.

```
student@bchd:~/mycode/flaskapi$ vim /home/student/sshpass.yml
```

27. Create the following file using the YAML format containing three IP addresses, usernames, and passwords:

```
---
-
  ip: 10.10.2.3
  un: bender
  passw: alta3
-
  ip: 10.10.2.4
  un: fry
  passw: alta3
...
```

28. Press **Esc** and then save and exit with `:wq`

29. Now create your Flask application.

```
student@bchd:~/mycode/flaskapi$ vim graphapp01.py
```

30. Make the following script.

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   A flask app that returns visual graphs of server up times using matplotlib"""

# Python standard library
import re  # regex

# python3 -m pip install numpy
import numpy as np # number operations

# python3 -m pip install pyyaml
import yaml # pyyaml for yaml

# python3 -m pip install paramiko
import paramiko # ssh into servers

# python3 -m pip install flask
from flask import Flask, render_template

# python3 -m pip install matplotlib
import matplotlib.pyplot as plt


# a function that performs an SSH operation to remote systems
def sshlogin(ip, un, passw):
    sshsession = paramiko.SSHClient()
    sshsession.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    sshsession.connect(hostname=ip, username=un, password=passw)
    ssh_stdin, ssh_stdout, ssh_stderr = sshsession.exec_command("cat /proc/uptime")
    sshresult = ssh_stdout.read().decode('utf-8').split()[0]
    with open("sshresult", "w") as myfile:
        myfile.write(sshresult)
    days = (int(float(sshresult)) / 86400)  # convert uptime in sec to days
    sshsession.close()
    print(days)
    return days

app = Flask(__name__)

@app.route("/graphin")
def graphin():
    with open("/home/student/sshpass.yml") as sshpass: # creds for our servers
        creds = yaml.load(sshpass)
    svruptime = []
    xtick = []
    for cred in creds:
        xtick.append(cred['ip'])
        resp = sshlogin(cred['ip'], cred['un'], cred['passw'])
        svruptime.append(resp)
    xtick = tuple(xtick) # create a tuple
    svruptime = tuple(svruptime)

    # graphin
    N = 2 # total number of bars
    ind = np.arange(N)    # the x locations for the groups
    width = 0.35       # the width of the bars: can also be len(x) sequence
    p1 = plt.bar(ind, svruptime, width)

    plt.ylabel('Uptime in Days')

    plt.title('Uptime of Servers in Days')
    plt.xticks(ind, xtick)
    plt.yticks(np.arange(0, 20, 1)) # prob want to turn this into a log scale

    plt.savefig('static/status.png') # might want to save this with timestamp for history purposes
    return render_template("graph.html")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=2224)
```

31. Press **Esc** and then save and exit with `:wq`

32. Create `templates/graph.html`

```
student@bchd:~/mycode/flaskapi$ vim templates/graph.html
```

33. Create the template to continually return. Note that the meta-tag is responsible for refreshing the page (and thus recalling the Python code) every 30 seconds!

```
<html>
<head>
    <meta http-equiv="refresh" content="30">
</head>
<body>
    <img src="/static/status.png" alt="User Image">
</body>
</html>
```

34. Press **Esc** and then save and exit with `:wq`

35. Run your new Flask application:

```
student@bchd:~/mycode/flaskapi$ python3 graphapp01.py
```

36. With your application running, open `aux1` on your `Group Dashboard` tab. Add `/graphin` to the end of the URL in the browser tab that opens.

37. **CHALLENGE 01 (OPTIONAL)** - Rewrite to code to support a 3rd server: zoidberg@10.10.2.5 (pw: alta3)

38. **CHALLENGE 02 (OPTIONAL)** - Rewrite to code to support "n" servers (return a graph built dynamically). A 4th server is available farnsworth@10.10.2.6 (pw: alta3)

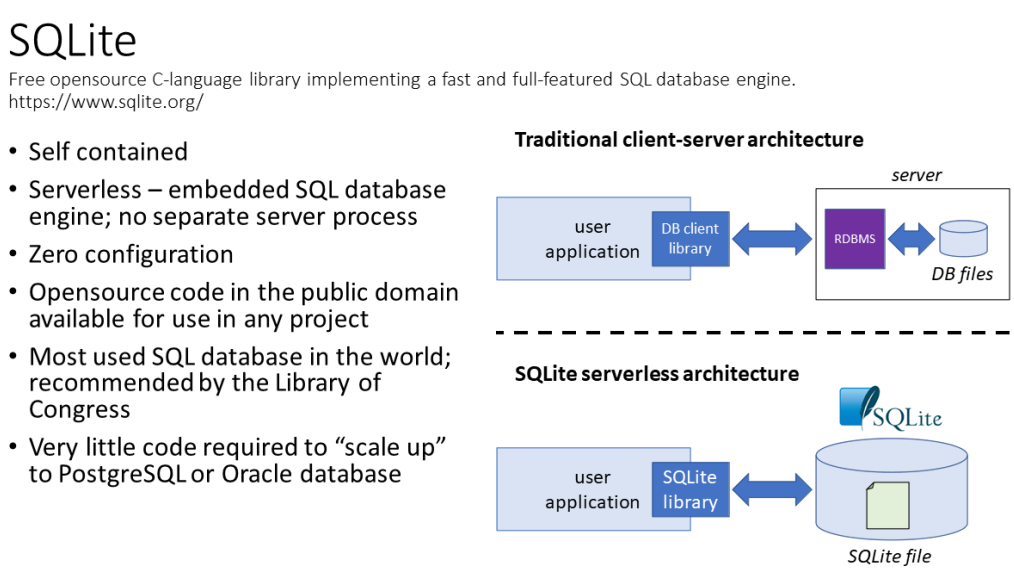
39. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "flask and upload script"`
    - `git push origin`

43. LECTURE - Learning sqlite3

Lab Objective

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.



The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249.

The library official documentation is available here: https://docs.python.org/3/library/sqlite3.html

Procedure

1. Create a new directory to work in, `~/mycode/sqldb/`

   ```
   student@bchd:~$ mkdir /home/student/mycode/sqldb
   ```

2. Move into your new directory.

   ```
   student@bchd:~$ cd /home/student/mycode/sqldb
   ```

3. Create a new script, `database01.py`. This script will create a database called `test.db`.

   ```
   student@bchd:~/mycode/sqldb$ vim database01.py
   ```

4. Following Python code shows how to connect to an existing database. If the database does not exist, then it will be created and finally a database object will be returned. Copy and paste the following into your script.

   ```
   #!/usr/bin/python3

   import sqlite3
   conn = sqlite3.connect('test.db')
   print("Opened database successfully")
   ```

5. Save and exit with `:wq`

6. Run your script.

   ```
   student@bchd:~/mycode/sqldb$ python3 database01.py
   ```

7. Create a new script, `database02.py`. This script will create a table called `company` within our database.

   ```
   student@bchd:~/mycode/sqldb$ vim database02.py
   ```

8. Copy and paste the following into your script.

```
#!/usr/bin/env python3

import sqlite3
conn = sqlite3.connect('test.db')
print("Opened database successfully")
conn.execute('''CREATE TABLE COMPANY
 (ID INT PRIMARY KEY     NOT NULL,
 NAME           TEXT    NOT NULL,
 AGE            INT     NOT NULL,
 ADDRESS        CHAR(50),
 SALARY         REAL);''')
print("Table created successfully")
conn.close()
```

9. Save and exit with `:wq`

10. Run your script.

```
student@bchd:~/mycode/sqldb$ python3 database02.py
```

11. Run your script a second time... this time it will **fail**

```
student@bchd:~/mycode/sqldb$ python3 database02.py
```

12. The script failed because your table already exists! Create a new script that will not have this issue.

```
student@bchd:~/mycode/sqldb$ vim database02v2.py
```

13. Rather than try to handle the error with Python, we can improve the way we create our table in sqlite. By changing the language to `CREATE TABLE IF NOT EXISTS COMPANY`, we can prevent the error from ever occurring. Create the following *improved* script.

```
#!/usr/bin/env python3

import sqlite3
conn = sqlite3.connect('test.db')
print("Opened database successfully")
conn.execute('''CREATE TABLE IF NOT EXISTS COMPANY
 (ID INT PRIMARY KEY     NOT NULL,
 NAME           TEXT    NOT NULL,
 AGE            INT     NOT NULL,
 ADDRESS        CHAR(50),
 SALARY         REAL);''')
print("Table created successfully")
conn.close()
```

14. Save and exit with `:wq`

15. Run your new script. It should no longer cause an error.

```
student@bchd:~/mycode/sqldb$ python3 database02v2.py
```

16. Run your script a second time, just to make sure we no longer are throwing errors.

```
student@bchd:~/mycode/sqldb$ python3 database02v2.py
```

17. Write a script that will place some data into our database. Create `database03.py`.

```
student@bchd:~/mycode/sqldb$ vim database03.py
```

18. Copy and paste the following into your script.

```
#!/usr/bin/env python3

import sqlite3
conn = sqlite3.connect('test.db')
print("Opened database successfully")

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (1, 'Paul', 32, 'California', 20000.00 )")

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (2, 'Allen', 25, 'Texas', 15000.00 )")

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )")

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )")

conn.commit()
print("Records created successfully")
conn.close()
```

19. Save and exit with `:wq`

20. Run your script.

```
student@bchd:~/mycode/sqldb$ python3 database03.py
```

21. Create a new script, `database04.py`. This script will select some of our data from the database and print it out.

```
student@bchd:~/mycode/sqldb$ vim database04.py
```

22. Copy and paste the following into your script.

```
#!/usr/bin/env python3

import sqlite3
conn = sqlite3.connect('test.db')
print("Opened database successfully")
cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
    print("ID = ", row[0])
    print("NAME = ", row[1])
    print("ADDRESS = ", row[2])
    print("SALARY = ", row[3], "\n")

print("Operation done successfully")
conn.close()
```

23. Save and exit with `:wq`

24. Run your script.

```
student@bchd:~/mycode/sqldb$ python3 database04.py
```

25. Create a new script, `database05.py`. This script will update some of our data from the database and print it out.

```
student@bchd:~/mycode/sqldb$ vim database05.py
```

26. Copy and paste the following into your script.

```
#!/usr/bin/env python3

import sqlite3

conn = sqlite3.connect('test.db')
print("Opened database successfully")

conn.execute("UPDATE COMPANY set SALARY = 25000.00 where ID = 1")
conn.commit()
print("Total number of rows updated :", conn.total_changes)

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
    print("ID = ", row[0])
    print("NAME = ", row[1])
    print("ADDRESS = ", row[2])
    print("SALARY = ", row[3], "\n")

print("Operation done successfully")
conn.close()
```

27. Save and exit with `:wq`

28. Run your script.

```
student@bchd:~/mycode/sqldb$ python3 database05.py
```

29. Create a new script, `database06.py`. This script will delete some of our data from the database and print out the modified database.

```
student@bchd:~/mycode/sqldb$ vim database06.py
```

30. Copy and paste the following into your script.

```
#!/usr/bin/env python3

import sqlite3

conn = sqlite3.connect('test.db')
print("Opened database successfully")

conn.execute("DELETE from COMPANY where ID = 2;")
conn.commit()
print("Total number of rows deleted :", conn.total_changes)

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
    print("ID = ", row[0])
    print("NAME = ", row[1])
    print("ADDRESS = ", row[2])
    print("SALARY = ", row[3], "\n")

print("Operation done successfully")
conn.close()
```

31. Save and exit with `:wq`
32. Run your script.

```
student@bchd:~/mycode/sqldb$ python3 database06.py
```

33. Be sure to check up on Python sqlite3 module's official documentation: https://docs.python.org/3/library/sqlite3.html
34. If you're tracking your code in a SCM, issue the following commands:
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "sql operations and python3"`
    - `git push origin`

44. Tracking API Data with sqlite3

Lab Objective

The objective of this lab is to create a practical application that can harness data from an API and then save it within a SQL database for long term storage. This application is written to look up data from the Open Movies DataBase (OMDB) API, but could easily be modified to look up data from other API sources.

Procedure

1. In this lab, we'll use the Open Movie Data Base API to grab some metadata about movies to write into our local SQL database. Create a new directory to work in, `~/mycode/apisqlite/`

   ```
   student@bchd:~$ mkdir /home/student/mycode/apisqlite
   ```

2. Move into your new directory.

   ```
   student@bchd:~$ cd /home/student/mycode/apisqlite
   ```

3. To have success with this lab, you'll need to go to https://www.omdbapi.com/apikey.aspx and sign up for an API key. Be sure to only sign up for a **FREE** account.

4. You'll need to confirm your API key via an email. Click the link at the bottom of the email to confirm your key.

5. Create a file to store your key in.

   ```
   student@bchd:~/mycode/apisqlite$ vim ~/omdb.key
   ```

6. Copy your key out of your email, and paste it into this file. Your key is only 8 characters long, and should be at the top of the email itself. When you paste it into this file, be sure you didn't include any extra lines or whitespaces.

7. Save and exit with `:wq`

8. Create a new python script.

   ```
   student@bchd:~/mycode/apisqlite$ vim apisqlite01.py
   ```

9. Copy and paste the following into your new script:

```
#!/usr/bin/env python3
""" Author: RZFeeser || Alta3 Research
Gather data returned by various APIs published on OMDB, and cache in a local SQLite DB
"""

import json
import sqlite3
import requests

# Define the base URL
OMDBURL = "http://www.omdbapi.com/?"

# search for all movies containing string
def movielookup(mykey, searchstring):
    """Interactions with OMDB API
       mykey = omdb api key
       searchstring = string to search for"""
    try:
        # begin constructing API
        api = f"{OMDBURL}apikey={mykey}&s={searchstring}"

        ## open URL to return 200 response
        resp = requests.get(api)
        ## read the file-like object decode JSON to Python data structure
        return resp.json()
    except:
        return False

def trackmeplease(datatotrack):
    conn = sqlite3.connect('mymovie.db')
    try:
        conn.execute('''CREATE TABLE IF NOT EXISTS MOVIES (TITLE TEXT PRIMARY KEY NOT NULL, YEAR INT  NOT NULL);''')

        # loop through the list of movies that was passed in
        for data in datatotrack:
            # in the line below, the ? are examples of "bind vars"
            # this is best practice, and prevents sql injection attacks
            # never ever use f-strings or concatenate (+) to build your executions
            conn.execute("INSERT INTO MOVIES (TITLE,YEAR) VALUES (?,?)",(data.get("Title"), data.get("Year")))
            conn.commit()

        print("Database operation done")
        conn.close()
        return True
    except:
        return False

# Read in API key for OMDB
def harvestkey():
    with open("/home/student/omdb.key") as apikeyfile:
        return apikeyfile.read().rstrip("\n") # grab the api key out of omdb.key

def printlocaldb():
    pass
    #cursor = conn.execute("SELECT * from MOVIES")
    #for row in cursor:
    #    print("MOVIE = ", row[0])
    #    print("YEAR = ", row[1])


def main():

    # read the API key out of a file in the home directory
    mykey = harvestkey()

    # enter a loop condition with menu prompting
    while True:
        # initialize answer
        answer = ""
        while answer == "":
            print("""\n**** Welcome to the OMDB Movie Client and DB ****
            ** Returned data will be written into the local database **
            1) Search for All Movies Containing String
            2) Search for Movies Containing String, and by Type
            99) Exit""")

            answer = input("> ") # collect an answer for testing

        # testing the answer
        if answer in ["1", "2"]:
            # All searches require a string to include in the search
            searchstring = input("Search all movies in the OMDB. Enter search string: ")

            if answer == "1":
```

```
                resp = movielookup(mykey, searchstring)
            elif answer == "2":
                print("\nSearch by type coming soon!\n") # maybe you can write this code!
                continue                                # restart the while loop
            if resp:
                # display the results
                resp = resp.get("Search")
                print(resp)
                # write the results into the database
                trackmeplease(resp)
            else:
                print("That search did not return any results.")

        # user wants to exit
        elif answer == "99":
            print("See you next time!")
            break

if __name__ == "__main__":
    main()
```

10. Save and exit with `:wq`
11. Run your code.

```
student@bchd:~/mycode/apisqlite$ python3 apisqlite01.py
```

12. Try running the script a few times. Search for a few different movies.
13. Ensure the sqlite3 client is installed with `apt`

```
student@bchd:~/mycode/apisqlite$ sudo apt install sqlite3
```

14. Connect to the SQL database client. Information on the client can be found here: https://sqlite.org/cli.html *Note: If you get stuck in this client, press CTRL + D*

```
student@bchd:~/mycode/apisqlite$ sqlite3
```

15. Open your database file, `mymovie.db`

```
sqlite> .open mymovie.db
```

16. Get the tables within `mymovie.db`

```
sqlite> .tables
```

17. Select all of the data in the table, and display it on the screen.

```
sqlite> .dump
```

18. Exit the SQL database client.

```
sqlite> .quit
```

19. **CUSTOMIZATION 01** - After reviewing the API usage on https://www.omdbapi.com/. Add an "option 2" that allows the user to limit their search by "type".
20. **CUSTOMIZATION 02** - After reviewing the API usage on https://www.omdbapi.com/. Add an "option 3" that allows the user to limit their search by "year of release".
21. **CUSTOMIZATION 03** - After reviewing the API usage on https://www.omdbapi.com/. Add an "option 4" that allows the user to limit their search by "type" and "year of release".
22. *CUSTOMIZATION SOLUTION 01 to 03 available @ **https://static.alta3.com/courses/pyapi/apisqlite02.py**
23. **CODE CUSTOMIZATION 04** - Add an "option 5" that displays the contents of the LOCAL database.
24. **CODE CUSTOMIZATION 05** - Replace the CLI User Interface with a User Interface provided by Flask. All of the same "options" should be available via HTTP GET's to your local Flask server.
25. If you're tracking your code in a SCM, issue the following commands:
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "Learning to cache data locally pulled from APIs"`
    - `git push origin`

45. Tracking Inventory with sqlite

Lab Objective

We're going to create two applications: a flask API that manipulates a SQL database, as well as a client to speak to our API. Our application is going to track a collection of student information, but could easily be manipulated to track any type of collection.

Be sure to check up on Python sqlite3 module's official documentation: https://docs.python.org/3/library/sqlite3.html

Procedure

1. Create a new directory to work in and a place to stash templates: `~/mycode/myapp/templates/`

   ```
   student@bchd:~$ mkdir -p /home/student/mycode/myapp/templates/
   ```

2. Move into your new directory.

   ```
   student@bchd:~$ cd /home/student/mycode/myapp
   ```

3. Create a new script, `server01.py`. This script will handle our server-side logic: hosting the Flask API, interacting with and connecting to the sqlite database.

   ```
   student@bchd:~/mycode/myapp$ vim server01.py
   ```

4. Copy and paste the following into your script.

```
#!/usr/bin/python3
"""RZFeeser || Alta3 Research
Tracking student inventory within a sqliteDB accessed
via Flask APIs"""

# standard library
import sqlite3 as sql

# python3 -m pip install flask
from flask import Flask
from flask import render_template
from flask import request

app = Flask(__name__)

# return home.html (landing page)
@app.route('/')
def home():
    return render_template('home.html')

# return student.html (a way to add a student to our sqliteDB)
@app.route('/enternew')
def new_student():
    return render_template('student.html')

# if someone uses student.html it will generate a POST
# this post will be sent to /addrec
# where the information will be added to the sqliteDB
@app.route('/addrec',methods = ['POST'])
def addrec():
    try:
        nm = request.form['nm']         # student name
        addr = request.form['addr']     # student street address
        city = request.form['city']     # student city
        pin = request.form['pin']       # "pin" assigned to student
                                        # ("pin" is just an example of meta data we want to track)

        # connect to sqliteDB
        with sql.connect("database.db") as con:
            cur = con.cursor()

            # place the info from our form into the sqliteDB
            cur.execute("INSERT INTO students (name,addr,city,pin) VALUES (?,?,?,?)",(nm,addr,city,pin) )
            # commit the transaction to our sqliteDB
            con.commit()
        # if we have made it this far, the record was successfully added to the DB
        msg = "Record successfully added"

    except:
        con.rollback()  # this is the opposite of a commit()
        msg = "error in insert operation"    # we were NOT successful

    finally:
        con.close()     # successful or not, close the connection to sqliteDB
        return render_template("result.html",msg = msg)    #

# return all entries from our sqliteDB as HTML
@app.route('/list')
def list_students():
    con = sql.connect("database.db")
    con.row_factory = sql.Row

    cur = con.cursor()
    cur.execute("SELECT * from students")           # pull all information from the table "students"

    rows = cur.fetchall()
    return render_template("list.html",rows = rows) # return all of the sqliteDB info as HTML

if __name__ == '__main__':
    try:
        # ensure the sqliteDB is created
        con = sql.connect('database.db')
        print("Opened database successfully")
        # ensure that the table students is ready to be written to
        con.execute('CREATE TABLE IF NOT EXISTS students (name TEXT, addr TEXT, city TEXT, pin TEXT)')
        print("Table created successfully")
        con.close()
        # begin Flask Application
        app.run(host="0.0.0.0", port=2224, debug = True)
    except:
        print("App failed on boot")
```

5. Save and exit with `:wq`

6. Create an HTML script for `student.html`

```
student@bchd:~/mycode/myapp$ vim templates/student.html
```

7. Copy and paste the following into your new script. As it can be seen, form data is posted to the `/addrec` URL which binds the addrec() function.

```
<html>
   <body>
      <form action = "{{ url_for('addrec') }}" method = "POST">
         <h3>Student Information</h3>
         Name:<br>
         <input type = "text" name = "nm" /><br>

         Address:<br>
         <textarea name = "addr" ></textarea><br>

         City:<br>
         <input type = "text" name = "city" /><br>

         PINCODE<br>
         <input type = "text" name = "pin" /><br>
         <input type = "submit" value = "submit" /><br>
      </form>
   </body>
</html>
```

8. Save and exit with `:wq`

9. In our script, notice that the **addrec()** function retrieves the form's data via a POST method and inserts in the table. A message corresponding to a success or error is rendered to `result.html`. We should create the script `result.html` now.

```
student@bchd:~/mycode/myapp$ vim templates/result.html
```

10. Copy and paste the following into the new script. The escaping statement (msg) should display the result of the insert operation.

```
<!doctype html>
<html>
   <body>
      result of addition : {{ msg }}
      <h2><a href = "\">go back to home page</a></h2>
   </body>
</html>
```

11. Save and exit with `:wq`

12. We'll need `list.html`, another template, which iterates over the row set and renders the data in an HTML table.

```
student@bchd:~/mycode/myapp$ vim templates/list.html
```

```
<!doctype html>
<html>
   <body>
      <table border = 1>
         <thead>
            <td>Name</td>
            <td>Address</td>
            <td>City</td>
            <td>Pincode</td>
         </thead>

         {% for row in rows %}
            <tr>
               <td>{{ row["name"] }}</td>
               <td>{{ row["addr"] }}</td>
               <td>{{ row["city"] }}</td>
               <td>{{ row["pin"] }}</td>
            </tr>
         {% endfor %}
      </table>

      <a href = "/">Go back to home page</a>
   </body>
</html>
```

13. Save and exit with `:wq`

14. At one point we need to render a `home.html` which acts as the entry point of the application.

```
student@bchd:~/mycode/myapp$ vim templates/home.html
```

15. Copy and paste the following into your new script.

```
<!doctype html>
<html>
   <body>
      <h2>Welcome to my Flask & database app</h2>
      <a href = "/">Home Page (you are here)</a>
      <a href = "/enternew">Make a New Database Entry</a>
      <a href = "/list">View Records in the Database</a>
   </body>
</html>
```

16. Try running your script!

```
student@bchd:~/mycode/myapp$ python3 server01.py
```

17. Create a new record or two. Fill the appropriate form fields and submit it. The underlying function inserts the record in the students table. Click 'Show List' link to see the current data within the sqlite database.

18. Back in your **SECOND TMUX PANE**, cURL the root of your API.

```
student@bchd:~$ curl http://127.0.0.1:2224/
```

19. We can make an addition by sending a POST to `/addrec`. Let's try that now.

```
student@bchd:~$ curl http://127.0.0.1:2224/addrec -d "nm=David" -d "addr=Texas" -d "city=Dallas" -d "pin=1"
```

20. Great! Looks like a user was successfully added! Let's now add a second one.

```
student@bchd:~$ curl http://127.0.0.1:2224/addrec -d "nm=Tailor" -d "addr=FL" -d "city=Orlando" -d "pin=2"
```

21. Try getting everything out of the students table within the database. The result of this cURL command will be a an HTML page dynamically populated students from the sqlite database.

```
student@bchd:~$ curl http://127.0.0.1:2224/list
```

22. **CHALLENGE 01**: Try to make an entry and commit it to the database. Then pull your results.

23. **CHALLENGE 02** - Add functionality to the script, so that a student may be REMOVED from the database.

24. *SOLUTION 02 - A possible solution to CHALLENGE 02 is as follows:*

```
#!/usr/bin/python3
'''RZFeeser || Alta3 Research
SOLUTION 02 - Adding the ability to REMOVE data from the database with:

curl -X DELETE "localhost:2224/remove?name=Timmy"
curl -X DELETE "localhost:2224/remove?name=Jane"
curl -X DELETE "localhost:2224/remove?name=Larry"
'''

# standard library
import sqlite3 as sql

# python3 -m pip install flask
from flask import Flask
from flask import render_template
from flask import request

app = Flask(__name__)

# return home.html (landing page)
@app.route('/')
def home():
    return render_template('home.html')

# return student.html (a way to add a student to our sqliteDB)
@app.route('/enternew')
def new_student():
    return render_template('student.html')

# if someone uses student.html it will generate a POST
# this post will be sent to /addrec
# where the information will be added to the sqliteDB
@app.route('/addrec',methods = ['POST'])
def addrec():
    try:
        nm = request.form['nm']         # student name
        addr = request.form['addr']     # student street address
        city = request.form['city']     # student city
        pin = request.form['pin']       # "pin" assigned to student
                                        # ("pin" is just an example of meta data we want to track)

        # connect to sqliteDB
        with sql.connect("database.db") as con:
            cur = con.cursor()

            # place the info from our form into the sqliteDB
            cur.execute("INSERT INTO students (name,addr,city,pin) VALUES (?,?,?,?)",(nm,addr,city,pin) )
            # commit the transaction to our sqliteDB
            con.commit()
        # if we have made it this far, the record was successfully added to the DB
        msg = "Record successfully added"

    except:
        con.rollback()  # this is the opposite of a commit()
        msg = "error in insert operation"    # we were NOT successful

    finally:
        return render_template("result.html",msg = msg)    #

# return all entries from our sqliteDB as HTML
@app.route('/list')
def list_students():
    con = sql.connect("database.db")
    con.row_factory = sql.Row

    cur = con.cursor()
    cur.execute("SELECT * from students")           # pull all information from the table "students"

    rows = cur.fetchall()

    return render_template("list.html",rows = rows) # return all of the sqliteDB info as HTML

# use a HTTP DELETE to remove an entry from the table
@app.route('/remove', methods = ['DELETE'])
def remove():
    try:  # HTTP DELETE arrives at /remove?name=<name in DB to remove>

        name_to_remove = request.args.get("name") # peel off arguments and capture name to be removed

        with sql.connect("database.db") as con:
            cur = con.cursor()
            cur.execute("SELECT * FROM students WHERE name=(?)",(name_to_remove,))
```

```
            data = cur.fetchall()
            if len(data) == 0:
                msg = "record does not exist"
            else:
                # place the info from our form into the sqliteDB
                cur.execute("DELETE FROM students WHERE name=(?)",(name_to_remove,) )

                # commit the transaction to our sqliteDB
                con.commit()

                # if we have made it this far, the record was successfully added to the DB
                msg = "record successfully removed"

    except:
        msg = "error in removing the record"
    finally:
        return render_template("result.html",msg = msg) # return success

if __name__ == '__main__':
    try:
        # ensure the sqliteDB is created
        con = sql.connect('database.db')
        print("Opened database successfully")
        # ensure that the table students is ready to be written to
        con.execute('CREATE TABLE IF NOT EXISTS students (name TEXT, addr TEXT, city TEXT, pin TEXT)')
        print("Table created successfully")
        con.close()
        # begin Flask Application
        app.run(host="0.0.0.0", port=2224, debug = True)
    except:
        print("App failed on boot")
```

25. **CHALLENGE 03** - Rewrite this script to do track data you might be interested in. This could be prices of things on the internet, an inventory of some collection, teams, resources, staff, or anything else you dream up. To be successful your script doesn't need to be as complex as the script we wrote in this lab.

26. If you're tracking your code in a SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "Learning about APIs and DBs"`
    - `git push origin`

46. Flask and waitress

Lab Objective

The objective of this lab is to learn how to run flask with a WSGI server, specifically, waitress. When running publicly rather than in development, you should not use the Flask built-in development server (`flask run` or `app.run()`). The development server is provided by Werkzeug for convenience, but is not designed to be particularly efficient, stable, or secure.

Waitress is a production-quality pure-Python WSGI server with very acceptable performance. It has no dependencies except ones which live in the Python standard library. It runs on CPython on Unix and Windows under Python 3.7+. It is also known to run on PyPy 3 (version 3.7 compatible python) on UNIX. It supports HTTP/1.0 and HTTP/1.1.

Other WSGI options exist. Flask has documentation on all recommended options

References: Flask Documentation - Deploying to Production

- Flask Documentation - Waitress
- PyPi.org - waitress
- waitress Documentation
- Flask Documentation - Gunicorn
- Gunicorn "Green Unicorn"
- Flask Documentation - uWSGI
- Flask Documentation - mod_wsgi

Procedure

1. Move into (or create) a new directory to work in, `/home/student/mycode/waitress/

```
student@bchd:~$ mkdir -p /home/student/mycode/waitress/
```

2. Move into your new directory

```
student@bchd:~$ cd ~/mycode/waitress/
```

3. Now create a script called `wzug.py`

```
student@bchd:~/mycode/waitress$ vim werkzeug_svr.py
```

4. Copy the following into your Python script, `werkzeug_svr.py`

```
#!/usr/bin/python3
"""Alta3 Research | RZFeeser
   Running a script with the werkzeug built in server"""

from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "<h1 style='color:red'> Alta3 simple server! </h1>"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=2224)
```

5. Save and exit with `:wq`

6. Run the server.

```
student@bchd:~/mycode/waitress$ python3 werkzeug_svr.py
```

7. Split the screen with `ctrl` + `b` and then `"`.

8. Ensure your server can be contacted.

```
student@bchd:~$ curl localhost:2224/
```

9. You should receive a response. Close the split screen by typing `exit`.

10. Stop your server with `ctrl` + `c`

11. Install waitress.

```
student@bchd:~/mycode/waitress$ python3 -m pip install waitress
```

12. Build a better server, `waitress_svr.py`

```
student@bchd:~/mycode/waitress$ vim waitress_svr.py
```

13. Create the following script that uses `waitress`

```
#!/usr/bin/python3
"""Alta3 Research | RZFeeser
   Running a script with the waitress production server"""

from flask import Flask
# waitress must be imported to replace werkzeug
from waitress import serve


app = Flask(__name__)

@app.route("/")
def hello():
    return "<h1 style='color:red'> Alta3 simple server! </h1>"

if __name__ == "__main__":
    #app.run(host="0.0.0.0", port=2224) # commented out
    # new serve() command is part of waitress
    # serve() syntax is here https://docs.pylonsproject.org/projects/waitress/en/latest/arguments.html#arguments
    serve(app, host='0.0.0.0', port=2224)
```

14. Save and exit with `:wq`

15. Run the server, *when this server runs, it is not overly verbose. Provided it does not return an error, you can assume it is running.*

```
student@bchd:~/mycode/waitress$ python3 waitress_svr.py
```

16. Split the screen with `ctrl` + `b` and then `"`.

17. Ensure your new server can be contacted.

```
student@bchd:~$ curl localhost:2224/
```

18. You should receive a response. Close the split screen by typing `exit`.

19. Stop your server with `ctrl` + `c`

20. If you're tracking your code in a SCM, issue the following commands:

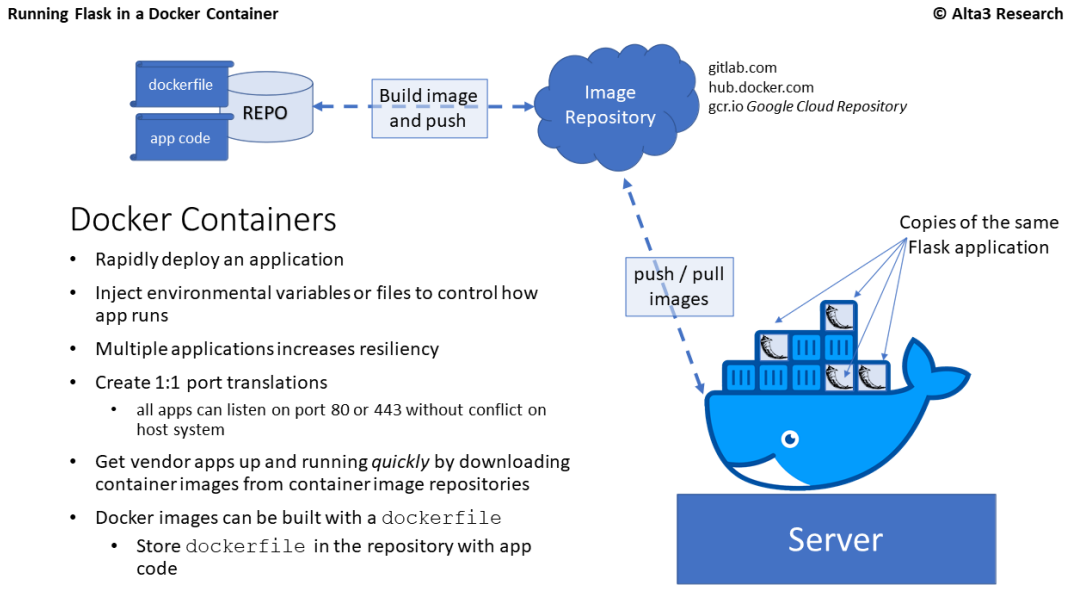
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "learning about flask waitress"`
    - `git push origin`

47. Running Flask in a Docker Container

Lab Objective

The objective of this lab is to give students an idea of how to run Flask application within a Docker container. Think of containers as light-weight virtualization (as opposed to a virtual machine).

Docker is easy to get started with. Most Linux operating systems install with, or give the option to install, Docker. Applications to run on your Docker platform can be found at: https://hub.docker.com



Procedure

1. Ensure you are in the home directory.

```
student@bchd:~$ cd
```

2. Confirm docker is installed by listing the various commands.

```
student@bchd:~$ docker --help
```

3. Imagine you're trying to deploy the following Python code, contained in `~/dune.py` The application is a simple "Hello Arrakis" app that uses Flask.

4. Create a directory to work in:

```
student@bchd:~$ mkdir ~/mycode/dune && cd ~/mycode/dune
```

5. Create the following file, `dune.py`

```
student@bchd:~/mycode/dune$ vim dune.py
```

```
#!/usr/bin/python3
"""By Chad Feeser | Alta3 Research
To use, try:
    curl localhost:5000/
    curl localhost:5000/atreides/
"""

from flask import Flask
app = Flask(__name__)

# if user sends HTTP GET to /
@app.route("/")
def index():
    return "In Frank Herbert's Dune, the Spice Melange makes space travel possible."

# if user sends HTTP GET to /atreides
@app.route("/atreides")
def atreides():
    return "As Dune opens, House Atreides is transitioning their rule to Arrakis, a desert planet."

# bind to all IP addresses port 5000
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

6. Exit with `:wq`

7. You'll need Flask to run this script, if you haven't installed it yet. Use `pip` to install `flask`.

```
student@bchd:~/mycode/dune$ python3 -m pip install flask
```

8. Great! Let's test our application.

```
student@bchd:~/mycode/dune$ python3 dune.py
```

9. Split your screen with **Ctrl** + **b** and then **Shift** + **"**

10. In the new tmux pane, curl your endpoints. The following command sends an HTTP GET to localhost:5000.

```
student@bchd:~$ curl http://localhost:5000/

In Frank Herbert's Dune, the Spice Melange makes space travel possible.
```

11. Great! Now send an HTTP GET to the second one.

```
student@bchd:~$ curl http://localhost:5000/atreides

As Dune opens, The House Atreides is transitioning their rule to Arrakis, a desert planet.
```

12. Looks like our Python Flask application is running just fine. Exit the current tmux pane.

```
student@bchd:~$ exit
```

13. Stop your Python Flask application by pressing **Ctrl** + **c**... if you leave it running it might attract sandworms!

14. To transform our application into a containerized app, we need to create a `Dockerfile`. The name `Dockerfile` is actually standardized, so don't try to rename it to something else. This file will tell Docker 'what to do' in order to create our container image.

```
student@bchd:~/mycode/dune$ vim Dockerfile

# This base image container is avail on hub.docker.com
# it has python 3.7 avail on Alpine Linux, a minimalist Linux distro
FROM python:alpine3.7
COPY . /app
WORKDIR /app
# Use Python package installer to install the Flask library to our image
RUN pip install -r requirements.txt
# container is exposed on port 5000
EXPOSE 5000
CMD python ./dune.py
```

15. Note that FROM directive is pointing to `python:alpine3.7`. This is telling Docker what base image to use for the container, and implicitly selecting what Python version to use, which in this case is 3.7. Docker Hub has base images for almost all supported versions of Python including 2.7. This example is using Python installed on Alpine Linux, a minimalist Linux distro, which helps keep the images small.

16. Also note the RUN directive that is calling PyPi (pip) and pointing to the `requirements.txt` file. This file contains a list of the dependencies that the application needs to run. Create that file now.

```
student@bchd:~/mycode/dune$ vim requirements.txt

# python dependencies
flask
```

17. The remaining directives in the Dockerfile are pretty straightforward. The CMD directive tells the container what to execute to start the application. In this case, it is telling Python to run `dune.py`. The COPY directive simply moves the application into the container image, WORKDIR sets the working directory, EXPOSE exposes a port that is used by Flask.

18. To build the image, run `docker build` from a command line or terminal that is in the root directory of the application. Including `-t` will "tag" our image as `dune-app`. Including `-f Dockerfile` instructs Docker to look to the file named `Dockerfile` for instructions on how to build the image.

```
student@bchd:~/mycode/dune$ sudo docker build -t dune-app -f Dockerfile .

...
Successfully built 4ffb50d3874c
Successfully tagged dune-app:latest
```

19. Now that our application is built, we can run the image, `dune-app`, within a Docker container named `scifi`.

```
student@bchd:~/mycode/dune$ sudo docker run --name scifi -d -p 5000:5000 dune-app
```

20. This starts the application as a container. The `--name` parameter names the container and the `-p` parameter maps the host's port 5000 to the container's port of 5000. Lastly, the `dune-app` is the image to run (we tagged it with this name). After it starts, you should be able to curl to the container. Try interacting with your endpoints as you did before.

```
student@bchd:~$ curl http://localhost:5000/

In Frank Herbert's Dune, the Spice Melange makes space travel possible.
```

21. Great! Now send an HTTP GET to the second one.

```
student@bchd:~$ curl http://localhost:5000/atreides

As Dune opens, The House Atreides is transitioning their rule to Arrakis, a desert planet.
```

22. We can look at the containers Docker is currently running with `docker ps`. Issuing this command should show the container `scifi` running the image `dune-app` on port `5000`.

```
student@bchd:~$ sudo docker ps

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
31bb91f27ecf        dune-app            "/bin/sh -c 'python …"   4 minutes ago       Up 4 minutes        0.0.0.0:5000->5000/tcp   scifi
```

23. Start ANOTHER instance of your app. The code has been "hardened" to listen on port `5000`. We may "only" have a single instance of port 5000 locally, but we can make Docker port forward **any** local port into the network namespace's port 5000 (where our new Flask application will be running). This time, use port `34727` (locally) and forward to port `5000` (within the container).

```
student@bchd:~/mycode/dune$ sudo docker run --name sandworm -d -p 34727:5000 dune-app
```

24. Look at that! One docker system running two applications, both using port 5000! Amazing.

25. Now try to `curl` your first Dune application (listening on local port 5000 and the container port 5000).

```
student@bchd:~$ curl http://localhost:5000/

In Frank Herbert's Dune, the Spice Melange makes space travel possible.
```

26. Great! Now send an HTTP GET to the second one (Docker knows to send local port 34727 to the container port 5000)

```
student@bchd:~$ curl http://localhost:34727/

In Frank Herbert's Dune, the Spice Melange makes space travel possible.
```

27. Both application should be sending back the same response. Imagine a load balancer in front of a bunch of applications, and you're starting to realize the power of containers and microservice architectures.

28. Because Docker is running our container, we can't 'just' issue a (CTRL + c) on our Flask app to stop it. Any command we would issue will result in Docker simply re-launching the container (Docker's job is to keep applications running). Therefore, we need to tell Docker to stop the container named `scifi`.

```
student@bchd:~$ sudo docker stop scifi
```

29. Now also stop `sandworm`

```
student@bchd:~$ sudo docker stop sandworm
```

30. Now that our image has been built, it has been cached by Docker locally. To run it, we no longer need to `docker build` the image. We can now simply deploy the built image. Where `dune-app` is the name of the image we created, docker will randomly spawn a name for our container.

```
student@bchd:~$ sudo docker run -d dune-app
```

31. Get some details about the container you just ran.

```
student@bchd:~$ sudo docker ps

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
0ee8c74c52eb        dune-app            "/bin/sh -c 'python …"   About a minute ago  Up About a minute   5000/tcp            friendly_heyrovsky
```

32. The only changes should be a new `CONTAINER ID`, as well as a new (randomly created) `NAMES`. In the previous example, we supplied the name via `--name scifi` for our container. This time, Docker created one randomly for us. The following command is a shortcut to stop all containers docker is currently running, regardless of how they are named (WARNING: this will also close Bender, Fry, Zoidberg, and Farnsworth. You'll need to rebuild those if you plan to use them again)

```
student@bchd:~$ sudo docker stop $(sudo docker ps -aq)
```

33. Naturally, more complex scenarios will require more attention to details, but the basic flow is the same for containerizing most applications.

34. Answer the following questions:

    - **Q1: How many times does Docker need to create the image?**
        - *A1: Just once. Building images takes time, to create the container image more than once is wasteful*
    - **Q2: Could the build process fail? Why or why not?**
        - *A2: Of course! In our example, an application dev could fail to produce the correct requirements.txt file, or point to the wrong base images to construct the image from, or any number of things!*
    - **Q3: Where does Docker fit into this?**
        - *A3: When code gets pushed into a repository, we want to turn it into an image and run it. Ideally, we want to isolate that code (application) and give it ONLY what it needs to do the job it was designed to do.*

35. **CHALLENGE 01 (OPTIONAL)**- An Alta3 instructor has put together a GitHub repository that will allow students to practice building API microservices with Flask and Docker containers: https://github.com/rzfeeser/simpleflaskservice

    - Your challenge is to clone this repository and instantiate a working Flask server (simpleflaskservice.py)
    - You can try this on your own or refer to the README.md inside the repository for help.
    - Build an image with this Flask app and run a container.
    - You'll know you have succeeded when you have the Flask server running inside of a Docker container and can curl the endpoints provided by this Flask application.

36. **CHALLENGE 02 (OPTIONAL)**- Write a Python client using the requests library to connect to your new microservices.

37. **CHALLENGE 03 (OPTIONAL)**- Add at least one more route that will return the *Dune* quotes in JSON format! This will require you to rebuild the image, give it a new tag.

48. LECTURE - Introduction to Django

Lab Objective

The objective of this lecture is to introduce Django.

Django Intro

Introduction to Django

What is django

- High-level Python web framework
- MVT
  - Model
  - View
  - Template
- ORM to support many common DBs
- Many popular sites use Django
- Lots of ready-to-use extras

Sites Built with Django
- Instagram
- Spotify
- YouTube
- The Washington Post
- BitBucket
- DropBox
- Mozilla
- EventBright
- Pinterest

The Django project is available at https://www.djangoproject.com/

Introduction to Django

Getting Started with Django

Required

- Python 3.8+ installed
  - Best to work in a virutalenv
- Browser
  - Chrome
  - Firefox
- Install Django 2.2+
  - `python3 -m pip install Django`
    - https://pypi.org/project/Django/

Helpful

- Your favorite IDE
  - Visual Studio Code
  - PyCharm
- Familiarity with HTTP
  - Requests
  - Responses
  - Packet captures (*.pcap)
    - Wireshark
    - tshark
    - tcpdump
    - termshark

All of these tools are provided for you within the Alta3 lab environment

Introduction to Django

Model, View, Template (MVT)

Model — Abstracting the database

- How to access and validate data
- Each attribute is a database field
- Fields are always required (INTEGER, VARCHAR, etc.)
- By writing python code you avoid having to know anything about the database itself (i.e. no SQL knowledge required)

View — How to respond to requests

- Accept HTTP requests and return HTTP responses like HTML documents
- Views usually appear in views.py of your application folder
- Described as functions or classes that receive a web request and return a web response
- Describes the model to use, the template to reach for, and other settings to tie together M&T

Template — Allows the creation of dynamic content

- "Fill in the blank"
- Create HTML, CSV, Configs, *anything with patterns*
- Jinja is based essentially Django templating with the addition of Jinja filters (Django does *not* have Jinja filters)

Introduction to Django

Models

- Contain data
  - How to access it
  - How to validate it
- Subclassed from `django.db.models.Model`
- Each attribute of the model is a DB field
- Django will auto-generate database-access API

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

provides

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Introduction to Django

Starting a Project

To start a project, run a command like the following:

```
$ django-admin startproject storefront .
```

- `django-admin` - tool installs with Django
- `startproject` - Keyword to create a new project
- `storefront` - Name of the project to be created
- `.` - Create project locally

Moraa Onwonga
moraa.onwonga@accenturefederal.com

Introduction to Django

Project Layout Basics

To start a project, run a command like the following:

```
$ django-admin startproject storefront .
```

Will create the following

```
|_ manage.py
|_ storefront/
   |_ __pycache__/
   |_ asgi.py
   |_ settings.py
   |_ urls.py
   |_ wsgi.py
```

- `manage.py` – Script to create apps, work with DBs, and start the dev web server
- `../asgi.py` – Standard that enables web apps to communicate. ASGI is the async standard that replaced sync WSGI.
- `../settings.py` – All website settings, registers applications, defines location of static files, DB config details, etc.
- `../urls.py` – The URL-to-view mappings of your project
- `../wsgi.py` – Helps Django communicate with the webserver

Your project is *ready to run!*

Introduction to Django

Start your server!

To test your project, you can use the "runserver" command:

```
$ python3 manage.py runserver
```

Will create the following

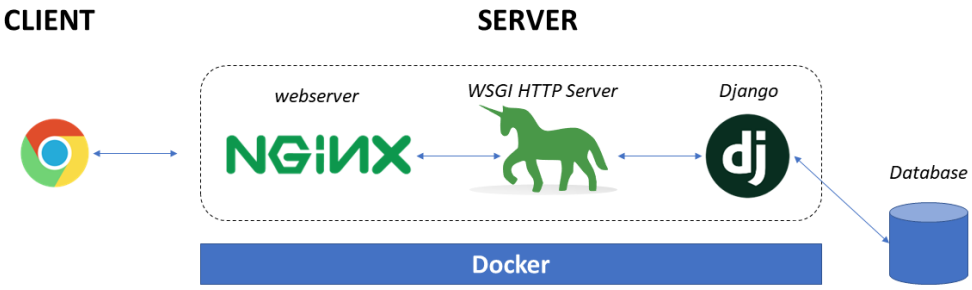
Test Server

Using the `runserver` command

- Comes with Django install
- Convenient
- Untested
- Slow
- Security concerns
- Docs are very adamant not to use this in a production setting

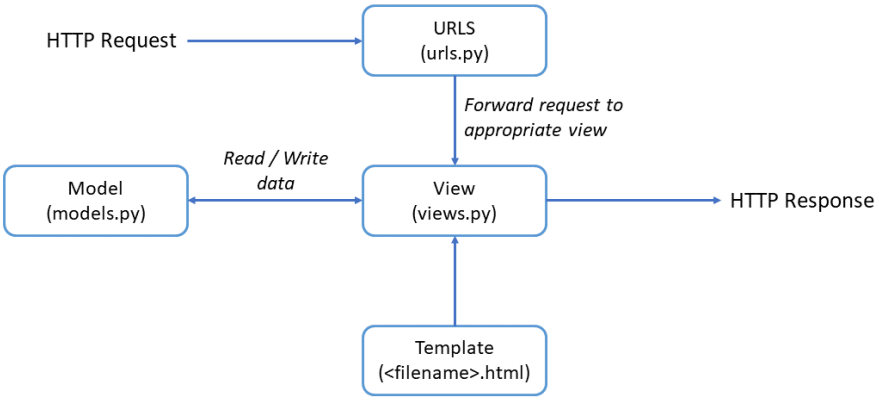
Introduction to Django

What does Production look like?



1. Client makes a request.
2. NGINX receives request. Configured to only let requests through that need to get through. Additionally offering: name routing, serving static files, handling many incoming requests, slow clients, termination of SSL (HTTPS happens here), forward requests that need to be dynamic to Gunicorn, caching, load balancing, and more.
3. Gunicorn translates the request into a format Django can understand (WSGI). Calls your code when requests come in. Translate WSGI response of your app into legal HTTP responses. Many choices besides Gunicorn.

MVT Relationship



49. Introduction to Django

Lab Objective

The objective of this lab is to provide a short overview of the web-framework, Django. Per the Django Project homepage, "Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app."

Django is a full-stack web framework for Python, whereas Flask is a lightweight and extensible Python web framework. Django's batteries-included approach enable programmers to accomplish common web development tasks without using third-party tools and libraries. Django makes it easier for **Django developers** to accomplish common web development tasks like user authentication, URL routing and database schema migration. Also, Django accelerates custom web application development by providing built-in template engine, ORM system, and bootstrapping tool. However, Django lacks some of the robust features provided by Python.

On the other hand, Flask is a lightweight, minimalist web framework. It lacks some of the built-in features provided by Django, but it helps **Python developers** to keep the core of a web application simple and extensible.

Still confused? Cars have automatic or standard transmissions. If you find the less-is-more approach to driving with an automatic transmission appealing, Django might be for you. If you feel more in control with a gearbox and clutch, than Flask might be your go to framework. There is not really a right or wrong argument to be had, as much as which solution works best for your brain.

Resources:

- Read more about the Django project here: https://www.djangoproject.com/
- Django uses a modeling style called MVT (Model, View, Template). Read more about these concepts here:
  - **Model**: https://docs.djangoproject.com/en/2.2/topics/templates/

Procedure

1. Within a terminal space, move into your home directory.

```
student@bchd:~$ cd
```

2. Make sure to install the virtual environment package.

```
student@bchd:~$ sudo apt install virtualenv -y
```

3. Create the virtual env and then activate it.

```
student@bchd:~$ python3 -m venv djintro
```

4. Start your virtual environment.

```
student@bchd:~$ source ./djintro/bin/activate
```

5. Install Django within the virtual environment.

```
(djintro) student@bchd:~$ pip install Django
```

6. Check your version using django-admin.

```
(djintro) student@bchd:~$ django-admin --version
```

7. The first time using Django, it is required to auto-generate some code that establishes a Django project. That's a bit different than the standard `import` approach.

8. A Django project is, effectively, a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings. This is completed with a `django-admin` utility. Type the following to create a new project `fifthelement`

```
(djintro) student@bchd:~$ django-admin startproject fifthelement
```

9. Locally, `fifthelement/` was created.

```
(djintro) student@bchd:~$ ls
```

10. Let us use `tree` to explore it. To be clear, `tree` has nothing to do with Django. It is just a tool for exploring directory structures.

```
(djintro) student@bchd:~$ sudo apt install tree -y
```

11. Run tree against your new directory.

```
(djintro) student@bchd:~$ tree fifthelement
```

```
fifthelement/
 - fifthelement
     - __init__.py
     - settings.py
     - urls.py
     - wsgi.py
 - manage.py
```

12. The files the tree utility exposed are:

    - **The outer** `fifthelement/` **root directory** is just a container for your project. Its name doesn't matter to Django; you can rename it to anything you like.
    - `manage.py` - A command-line utility that lets you interact with this Django project in various ways. You can read all the details about `manage.py` in `django-admin` and `manage.py`
    - **The inner** `fifthelement/` **directory** is the actual Python package for your project. Its name is the Python package name you'll need to use to import anything inside it (e.g. fifthelement.urls).
    - `fifthelement/__init__.py` - An empty file that tells Python that this directory should be considered a Python package. If you're a Python beginner, read more about packages in the official Python docs.
    - `fifthelement/settings.py` - Settings/configuration for this Django project. Django settings will tell you all about how settings work.
    - `fifthelement/urls.py` - The URL declarations for this Django project; a "table of contents" of your Django-powered site. You can read more about URLs in URL dispatcher.
    - `fifthelement/wsgi.py` - An entry-point for WSGI-compatible web servers to serve your project. See How to deploy with WSGI for more details.

13. Lets explore the development server we just spawned. Move into the directory, `fifthelement/`

```
(djintro) student@bchd:~$ cd fifthelement/
```

14. Establish a shell variable for the host we expect to send as a HTTP_HOST header to our Django server.

```
(acme_env) student@bchd:~/fifthelement$ AUX1=aux1-`hostname --domain`.live.alta3.com
```

15. Change the settings.py file to allow for your aux1 HTTP_HOST header.

```
(acme_env) student@bchd:~/fifthelement$ sed -i "s/ALLOWED_HOSTS = \[\]/ALLOWED_HOSTS = \['$AUX1', '127.0.0.1'\]/g" fifthelement/settings.py
```

16. Allow CSRF Tokens to trust your host as well.

```
(boxst) student@bchd:~/fifthelement$ echo "CSRF_TRUSTED_ORIGINS = ['https://$AUX1', 'http://127.0.0.1']" >> fifthelement/settings.py
```

17. Verify your Django project works. Change into the outer `fifthelement/` directory, then run the following command to launch your server.

```
(djintro) student@bchd:~/fifthelement$ python3 manage.py runserver
```

18. Split your screen with tmux with the following two commands

    - `CTRL` + `b` (press both keys at the same time)
    - `"` (you will need to use the `SHIFT` key)

19. In your new screen, try to curl against your webserver (by default Django runs on port 8000)

```
student@bchd:~$ curl http://127.0.0.1:8000/
```

20. You should get back something like HTML data that includes the words, "Congratulations!" along with some other HTML data suitable for rendering in a browser. You can scroll up and down using **CTRL** + **b** and then **[**. Use `q` to quit the scrolling.

21. Close your split-screen by typing `exit` in your current split-screen.

```
student@bchd:~$ exit
```

22. Within the window hosting your Django server, stop the server with `CTRL` + `c`

23. If you want to change the server's port, pass it as a command-line argument. For instance, start on port 2224.

```
(djintro) student@bchd:~/fifthelement$ python3 manage.py runserver 2224
```

24. Notice the service is now listening on port 2224.

25. Stop your Django server with `CTRL` + `c`

26. If you want to change the server's IP, pass it along with the port. For example, to listen on all available public IPs (which is useful if you are running Vagrant or want to show off your work on other computers on the network), use:

```
(djintro) student@bchd:~/fifthelement$ python3 manage.py runserver 0:2224
```

27. Stop your Django server with `CTRL` + `c`

28. Let's try building a project from scratch. Create an app called, `altaapp01.py`

```
(djintro) student@bchd:~/fifthelement$ vim altaapp01.py
```

29. Place the following settings into the file.

```
#!/usr/bin/python3
"""Alta3 Research
All in one Django settings to drive Djanjo webserver"""

DEBUG = True                            # verbose output and prevent config of
                                        # ALLOWED_HOSTS variable

SECRET_KEY = 'Youcandanc3youcanjiv3'    # required for a django app

ROOT_URLCONF = __name__                 # The path to the URIs (called URLs here) with the 'views' in
                                        # the project. The var __name__ means "look in this file"

urlpatterns = []                        # urlpatterns usually defined by ROOT_URLCONF
```

30. Save and exit with `:wq`
31. Run the server

```
(djintro) student@bchd:~/fifthelement$ django-admin runserver --pythonpath=. --settings=altaapp01
```

32. Try testing via `curl` after splitting the screen.

```
student@bchd:~$ curl http://127.0.0.1:8000/
```

33. Close your second screen with `exit`
34. Stop the server with `CTRL` + `c`
35. Create a new app, `altaapp02.py`

```
(djintro) student@bchd:~/fifthelement$ vim altaapp02.py
```

36. Ensure the app looks like the following:

```
#!/usr/bin/python3
"""Alta3 Research
All in one Django settings to drive Djanjo webserver
Includes a simple home endpoint and a matching
'view' within urlpatterns"""

# python3 -m pip install django
from django.urls import re_path
from django.http import HttpResponse

DEBUG = True                            # verbose output and prevent config of
                                        # ALLOWED_HOSTS variable

SECRET_KEY = 'Youcandanc3youcanjiv3'    # required for a django app

ROOT_URLCONF = __name__                 # The path to the URIs (called URLs here) with the 'views' in
                                        # the project. The var __name__ means "look in this file"

def home(request):
    return HttpResponse('Welcome to the Alta3 App\'s Homepage!')   # send back an HTTP 200

# now we have urlpatterns to defined
# regex r'^$' matches "nothing" and points to the function "home"
urlpatterns = [
    re_path(r'^$', home),
]
```

37. Save and exit with `:wq`
38. Run the server

```
(djintro) student@bchd:~/fifthelement$ django-admin runserver --pythonpath=. --settings=altaapp02
```

39. Try testing via `curl` after splitting the screen.

```
student@bchd:~$ curl http://127.0.0.1:8000/
```

40. Note the response is what was defined within our `def home()` function.
41. Close your second screen with `exit`
42. Stop the server with `CTRL` + `c`
43. Create a new settings file, `altaapp03.py`

```
(djintro) student@bchd:~/fifthelement$ vim altaapp03.py
```

44. Create the following:

```
#!/usr/bin/python3
"""Alta3 Research
All in one Django settings to drive Djanjo webserver

Includes a simple home endpoint and a matching
'view' within urlpatterns

The color of the heading may be scraped via ?color=<color>
Example:
127.0.0.1:8000/?color=green
127.0.0.1:8000/?color=purple"""

# python3 -m pip install django
from django.urls import re_path
from django.http import HttpResponse

DEBUG = True                            # verbose output and prevent config of
                                        # ALLOWED_HOSTS variable

SECRET_KEY = 'Youcandanc3youcanjiv3'    # required for a django app

ROOT_URLCONF = __name__                 # The path to the URIs (called URLs here) with the 'views' in
                                        # the project. The var __name__ means "look in this file"

# return an HTML template
def home(request):
    """return a template to any lookups to home with optional color"""

    # NOTE: The following directly snags input directly from the URIs
    # user input should always be scrutinized to avoid security issues like XSS
    color = request.GET.get('color', '')   # scrape out the parameter from ?color=<color>

    return HttpResponse(
        '<h1 style="color:' + color + '">Welcome to the Alta3 App\'s Homepage!</h1>'
    )  # don't use user input like that in real projects!

urlpatterns = [
    re_path(r'^$', home),
]
```

45. Save and exit with `:wq`
46. Run the server

```
(djintro) student@bchd:~/fifthelement$ django-admin runserver --pythonpath=. --settings=altaapp03
```

47. Try testing via `curl` after splitting the screen.

```
student@bchd:~$ curl http://127.0.0.1:8000/?color=green
```

48. Note the response is what was defined within our `def home()` function, and has the variable from the URI stripped out and applied to the HTML template.

```
(djintro) student@bchd:~/fifthelement$ curl http://127.0.0.1:8000/?color=blue
```

49. Close your second screen with `exit`
50. Stop the server with `CTRL` + `c`
51. Jinja works with Django as well! Sort of. Jinja actually grew out of the "Django templating engine." Suffice to say if you know Jinja, you'll feel at home making templatings within Django.

```
(djintro) student@bchd:~/fifthelement$ vim altaapp04.py
```

52. Let's return an HTML template.

```
#!/usr/bin/python3
"""Alta3 Research
All in one Django settings to drive Djanjo webserver

Includes a simple home endpoint and a matching
'view' within urlpatterns

The color of the heading may be scraped via ?color=<color>
Example:
127.0.0.1:8000/?color=green
127.0.0.1:8000/?color=purple"""

# python3 -m pip install django
from django.urls import re_path
from django.http import HttpResponse

DEBUG = True                            # verbose output and prevent config of
                                        # ALLOWED_HOSTS variable

SECRET_KEY = 'Youcandanc3youcanjiv3'    # required for a django app

ROOT_URLCONF = __name__                 # The path to the URIs (called URLs here) with the 'views' in
                                        # the project. The var __name__ means "look in this file"

# engage django templating engine (more a less, jinja)
TEMPLATES = [{'BACKEND': 'django.template.backends.django.DjangoTemplates'},]

# return an HTML template
def home(request):
    """return a template to any lookups to home with optional color"""

    # NOTE: The following directly snags input directly from the URIs
    # user input should always be scrutinized to avoid security issues like XSS
    color = request.GET.get('color', '')   # scrape out the parameter from ?color=<color>

    return HttpResponse(
        '<h1 style="color:' + color + '">Welcome to the Alta3 App\'s Homepage!</h1>'
    )  # don't use user input like that in real projects!



from django.template import engines
from django.template.loader import render_to_string

def about(request):
    title = 'Alta3App'
    author = 'RZFeeser'

    # Django templating engine is very close to Jinja templating
    about_template = '''<!DOCTYPE html>
    <html>
    <head>
      <title>{{ title }}</title>
    </head>
    <body>
      <h1>About {{ title }}</h1>
      <p>This Website was developed by {{ author }}.</p>
      <p>Now using the Django's Template Engine.</p>
      <p><a href="{% url 'homepage' %}">Return to the homepage</a>.</p>
    </body>
    </html>
    '''

    django_engine = engines['django']
    template = django_engine.from_string(about_template)
    html = template.render({'title': title, 'author': author})

    return HttpResponse(html)

# match on "nothing" for the homepage
# math on "about/" for the about page
urlpatterns = [
    re_path(r'^$', home, name='homepage'),
    re_path(r'^about/$', about, name='aboutpage'),
]
```

53. Save and exit with `:wq`

54. Run the server

```
(djintro) student@bchd:~/fifthelement$ django-admin runserver --pythonpath=. --settings=altaapp04
```

55. Try testing via `curl` after splitting the screen.

```
student@bchd:~$ curl http://127.0.0.1:8000/?color=green
```

56. Try testing the new `/about` endpoint

```
student@bchd:~$ curl http://127.0.0.1:8000/about/
```

57. Close your second screen with `exit`

58. Stop the server with `CTRL` + `c`

59. Deactivate the virtualenv

```
(djintro) student@bchd:~/fifthelement$ deactivate
```

60. Consider yourself introduced to Django! If you are saving your code via an SCM, you can move your working folder into the `~/mycode` folder and then commit it:

    - `cd`
    - `mv ~/fifthelement ~/mycode/`
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "introduction to Django"`
    - `git push origin HEAD`

50. Intro to Django Views

Lab Objective

The objective of this lab is to learn to start building Django views and to work with urls. A view function, or view for short, is a Python function that takes a web request and returns a web response.

This response can be the HTML contents of a web page, or a redirect, or a 404 error, or an XML document, or an image or anything, really. The view itself contains whatever arbitrary logic is necessary to return that response. This code can live anywhere you want, as long as it's on your Python path. There's no other requirement. For the sake of putting the code somewhere, the convention is to put views in a file called `views.py`, placed in your project or application directory.

Procedure

1. Within a terminal space, move into your home directory.

   ```
   student@bchd:~$ cd
   ```

2. Make sure to install the virtual environment package.

   ```
   student@bchd:~$ sudo apt install virtualenv -y
   ```

3. Create the virtual env and then activate it.

   ```
   student@bchd:~$ python3 -m venv coffeeshop
   ```

4. Start your virtual environment.

   ```
   student@bchd:~$ source ./coffeeshop/bin/activate
   ```

5. Install Django within the virtual environment.

   ```
   (coffeeshop) student@bchd:~$ pip install Django
   ```

6. The next command creates a django project called phoenixcafe. It automatically creates some directories and files to check out.

   ```
   (coffeeshop) student@bchd:~$ django-admin startproject phoenixcafe
   ```

7. Now move into `~/phoenixcafe/`

   ```
   (coffeeshop) student@bchd:~$ cd ~/phoenixcafe/
   ```

8. Install `tree`.

   ```
   (coffeeshop) student@bchd:~/phoenixcafe$ sudo apt install tree -y
   ```

9. Run `tree` to show the directory layout.

   ```
   (coffeeshop) student@bchd:~/phoenixcafe$ tree
   ```

10. Notice that we have two folders named `phoenixcafe`. The outer is the project name. The inner one is our main application.

11. Create a **new** `~/phoenixcafe/phoenixcafe/views.py` for the project.

    ```
    (coffeeshop) student@bchd:~/phoenixcafe$ vim ~/phoenixcafe/phoenixcafe/views.py
    ```

12. Inside the `views.py` copy and paste the following:

    ```
    #!/usr/bin/python3

    # imports from Django
    from django.shortcuts import render
    from django.http import HttpResponse

    # This view will return "Welcome to the Phoenix Cafe!" as text
    def welcome(request):
        return HttpResponse("Welcome to the Phoenix Cafe!")
    ```

13. Save and exit with `:wq`

14. Take a look at the current copy of `~/phoenixcafe/phoenixcafe/urls.py`. This was created when we made our project. It gives access to the `/admin/` path, which we're not exploring right now.

    ```
    (coffeeshop) student@bchd:~/phoenixcafe$ cat ~/phoenixcafe/phoenixcafe/urls.py
    ```

15. Get rid of the default copy that was created for us.

    ```
    (coffeeshop) student@bchd:~/phoenixcafe$ rm ~/phoenixcafe/phoenixcafe/urls.py
    ```

16. Now open a new file called `urls.py`. Copy and paste the following into the script.

```
(coffeeshop) student@bchd:~/phoenixcafe$ vim ~/phoenixcafe/phoenixcafe/urls.py
```

17. Create the following:

```
#!/usr/bin/python3

# imports from Django
from django.contrib import admin
from django.urls import path
from .import views

urlpatterns = [
    path('welcome/', views.welcome),
]
```

18. Save and exit with `:wq`

19. Establish a shell variable for the host we expect to send as a HTTP_HOST header to our Django server.

```
(coffeeshop) student@bchd:~/phoenixcafe$ AUX1=aux1-`hostname --domain`.live.alta3.com
```

20. Change the settings.py file to allow for your aux1 HTTP_HOST header.

```
(coffeeshop) student@bchd:~/phoenixcafe$ sed -i "s/ALLOWED_HOSTS = \[\]/ALLOWED_HOSTS = \['$AUX1'\]/g" phoenixcafe/settings.py
```

21. Allow CSRF Tokens to trust your host as well.

```
(coffeeshop) student@bchd:~/phoenixcafe$ echo "CSRF_TRUSTED_ORIGINS = ['https://$AUX1']" >> phoenixcafe/settings.py
```

22. Start your server on port 2224.

```
(coffeeshop) student@bchd:~/phoenixcafe$ python3 manage.py runserver 0:2224
```

23. Switch over to your Aux1 view - you will have to add `/welcome/` to the end of the url. You should see the `Welcome to the Phoenix Cafe!` phrase.

24. Return to your terminal.

25. Stop the server with **CTRL** + **C**

26. **CHALLENGE 01** - Create a new *view* called, "sleepy". It should return "Z-z-z-z-z-z-z!" when called. The URL to trigger the view should be `/sleepy/`

27. Let's try creating a view that returns the current date and time. Edit

```
(coffeeshop) student@bchd:~/phoenixcafe$ vim ~/phoenixcafe/phoenixcafe/views.py
```

28. Update the code with the `import datetime` and the block at the bottom.

```
# NEW - standard library imports go on top
import datetime    # NEW

# imports from Django
from django.shortcuts import render
from django.http import HttpResponse

# This view will return "Welcome to the Phoenix Cafe!" as text
def welcome(request):
    return HttpResponse("Welcome to the Phoenix Cafe!")


# your custom code will be here


# NEW - new view that returns the current date and time
def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)  # we are not returning a static string
```

29. Save and exit with `:wq`

30. To display this view, we need to set up the URL dispatcher.

```
(coffeeshop) student@bchd:~/phoenixcafe$ vim ~/phoenixcafe/phoenixcafe/urls.py
```

31. Augment the list `urls.py` to look like the following (new code at the end)

```
#!/usr/bin/python3

# imports from Django
from django.contrib import admin
from django.urls import path
from .import views

urlpatterns = [
    path('welcome/', views.welcome),
    # your custom path will be here
]

# NEW - Add this to the end of your code
# clock/ is the path to trigger our function
urlpatterns += [path('clock/', views.current_datetime),]
```

32. Save and exit with `:wq`

33. Start your server.

```
(coffeeshop) student@bchd:~/phoenixcafe$ python3 manage.py runserver 0:2224
```

34. Switch over to your Aux1 view - you will have to add `/welcome/` to the end of the url. You should **still** see the `Welcome to the Phoenix Cafe!` phrase.

35. Now try changing the path to `/clock/`. It should return the current date and time. Refresh the page a few times to ensure the time is updating correctly.

36. Return to your terminal.

37. Stop the server with **CTRL** + **C**

38. **CHALLENGE 02** - Create your own view and corresponding URL dispatcher. This is an opportunity to flex your Python skills and creativity. Here are a few suggestions:

    - The `/rand/` endpoint that returns a random number
    - Visiting `/greetings/` returns, "Good morning!", "Good afternoon!", or "Good night!" depending on the time of day.
    - Create a `/cake/` endpoint that returns a recipe for a cake

39. Disconnect from the virtual environment.

```
(coffeeshop) student@bchd:~/phoenixcafe$ deactivate
```

40. Return to the home directory.

```
student@bchd:~/phoenixcafe$ cd
```

41. If you are saving your code via an SCM, ensure your code is within the `~/mycode` folder and then run the following command:

    - `cd`
    - `mv ~/phoenixcafe/ ~/mycode/`
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "views and urls"`
    - `git push origin`

51. Controlling HTTP Response Codes

Lab Objective

The objective of this lab is to learn to learn to control which HTTP response codes are returned.

Procedure

1. Within a terminal space, move into your home directory.

```
student@bchd:~$ cd
```

2. Make sure to install the virtual environment package.

```
student@bchd:~$ sudo apt install virtualenv -y
```

3. Create the virtual env and then activate it.

```
student@bchd:~$ python3 -m venv boxst
```

4. Start your virtual environment.

```
student@bchd:~$ source ./boxst/bin/activate
```

5. Install Django within the virtual environment.

```
(boxst) student@bchd:~$ pip install Django
```

6. The next command creates a django project called boxstore. It automatically creates some directories and files to check out.

```
(boxst) student@bchd:~$ django-admin startproject boxstore
```

7. Now move into `~/boxstore/`

```
(boxst) student@bchd:~$ cd ~/boxstore/
```

8. Install `tree`. This has nothing to do with Django, it just lets us see directories from the CLI more easily.

```
(boxst) student@bchd:~/boxstore$ sudo apt install tree -y
```

9. Run `tree` to show the directory layout.

```
(boxst) student@bchd:~/boxstore$ tree
```

10. Notice that we have two folders named `boxstore`. The outer is the project name. The inner one is our main application. *We can have multiple applications within the project.*

11. Create a **new** `~/boxstore/boxstore/views.py` for the project.

```
(boxst) student@bchd:~/boxstore$ vim ~/boxstore/boxstore/views.py
```

12. Inside the `views.py` copy and paste the following:

```
#!/usr/bin/python3

# imports from Django
from django.shortcuts import render
from django.http import HttpResponse
from django.http import HttpResponseNotFound

# This view will return a 404 response
def ierror(request):
    return HttpResponseNotFound("Page was not found")

def success(request):
    return HttpResponse("Page was found")
```

13. Save and exit with `:wq`

14. Take a look at the current copy of `~/boxstore/boxstore/urls.py`. This was created when we made our project. It gives access to the `/admin/` path, which we're not exploring right now.

```
(boxst) student@bchd:~/boxstore$ cat ~/boxstore/boxstore/urls.py
```

15. Get rid of the default copy that was created for us.

```
(boxst) student@bchd:~/boxstore$ rm ~/boxstore/boxstore/urls.py
```

16. Now open a new file called `urls.py`. Copy and paste the following into the script.

```
(boxst) student@bchd:~/boxstore$ vim ~/boxstore/boxstore/urls.py
```

17. Create the following:

```
#!/usr/bin/python3

# imports from Django
from django.contrib import admin
from django.urls import path
from .import views

urlpatterns = [
    path('ierror/', views.ierror),
    path('success/', views.success),
]
```

18. Save and exit with `:wq`

19. Establish a shell variable for the host we expect to send as a HTTP_HOST header to our Django server.

```
(boxst) student@bchd:~/boxstore$ AUX1=aux1-`hostname --domain`.live.alta3.com
```

20. Change the settings.py file to allow for your aux1 HTTP_HOST header.

```
(boxst) student@bchd:~/boxstore$ sed -i "s/ALLOWED_HOSTS = \[\]/ALLOWED_HOSTS = \['$AUX1', '127.0.0.1'\]/g" boxstore/settings.py
```

21. Allow CSRF Tokens to trust your host as well.

```
(boxst) student@bchd:~/boxstore$ echo "CSRF_TRUSTED_ORIGINS = ['https://$AUX1', 'http://127.0.0.1']" >> boxstore/settings.py
```

22. Start your server on port 2224.

```
(boxst) student@bchd:~/boxstore$ python3 manage.py runserver 0:2224
```

23. Split the screen with **CTRL** + **b** and then **SHIFT** + **"**

24. Curl the endpoint `/success/`. It should return a 200 per the debug in the top panel.

```
student@bchd:~$ curl http://127.0.0.1:2224/success/
```

25. Curl the endpoint `/success/` in verbose mode to see all of the headers.

```
student@bchd:~$ curl http://127.0.0.1:2224/success/ -v
```

26. If you want to scroll, press **CTRL** + **b** and then **[**. Use the arrow keys to scroll. Press `q` to exit the screen scroll.

27. Curl the endpoint `/ierror/`. It should return a 404 per the debug in the top panel.

```
student@bchd:~$ curl http://127.0.0.1:2224/ierror/
```

28. You can also perform the curl on `/ierror/` in verbose mode if you'd like.

29. Close the second panel with `exit`

```
student@bchd:~$ exit
```

30. Stop the Django test server with **CTRL** + **C**

31. Edit `~/boxstore/boxstore/views.py`

```
(boxst) student@bchd:~/boxstore$ vim ~/boxstore/boxstore/views.py
```

32. Add the following snippet to the bottom of your file.

```
def customheader(request):
    x = {}
    x['learning'] = 'Django' # string:string
    x['speed'] = 55          # string:integer

    return HttpResponse(headers=x)  # adds our custom headers to the response

def customcode(request):
    return HttpResponse("Working on that", status=201)  # return teh response code 201 "created"
```

33. Save and exit with `:wq`

34. Update `urls.py`. Copy and paste the following into the bottom of the script.

```
(boxst) student@bchd:~/boxstore$ vim ~/boxstore/boxstore/urls.py
```

35. Add the following to the bottom of the file.

```
# add this snippet to the bottom of ~/boxstore/boxstore/urls.py
urlpatterns += [
    path('header/', views.customheader),   # call our custom header
    path('created/', views.customcode),       # call our 201 response code
]
```

36. Save and exit with `:wq`
37. Start your server on port 2224.

```
(boxst) student@bchd:~/boxstore$ python3 manage.py runserver 0:2224
```

38. Split the screen with **CTRL** + **b** and then **SHIFT** + **"**
39. Curl the endpoint `/header/` in verbose mode. The 200 response should include our custom headers.

```
student@bchd:~$ curl http://127.0.0.1:2224/header/ -v
```

40. If you want to scroll, press **CTRL** + **b** and then **[**. Use the arrow keys to scroll. Press `q` to exit the screen scroll.
41. Curl the endpoint `/created/` to return the 201 response with the 15 character "Working on that".

```
student@bchd:~$ curl http://127.0.0.1:2224/created/
```

42. Close the second panel with `exit`

```
student@bchd:~$ exit
```

43. Stop the Django test server with **CTRL** + **C**
44. **CHALLENGE 01** - Write some code that returns a some HTTP response codes you find interesting. A list can be found by visiting the IETF RFC standard https://datatracker.ietf.org/doc/html/rfc7231#section-6 try to come up with at least two responses. You'll need to create a `path()` for your code as well within `urls.py`
45. **CHALLENGE 02** - Write some code that returns an HTTP response with a custom header or two. You'll need to create a `path()` for your code as well within `urls.py`
46. Disconnect from the virtual environment.

```
(boxst) student@bchd:~/boxstore$ deactivate
```

47. Return to the home directory.

```
(boxst) student@bchd:~/boxstore$ cd
```

48. If you are saving your code via an SCM, ensure your code is within the `~/mycode` folder and then run the following command:

    - `cd`
    - `mv ~/boxstore/ ~/mycode/`
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "http response codes"`
    - `git push origin`

52. Returning JSON with Django

Lab Objective

The objective of this lab is learn to attach JSON to the response returned by Django. Before Django version 1.7 there was a requirement to import the json library to return JSON. A response could then be created as follows:

```
import json
from django.http import HttpResponse

def return_json(request):
    response_data = {}
    response_data['result'] = 'error'
    response_data['message'] = 'Some error message'
    return HttpResponse(json.dumps(response_data), content_type="application/json")
```

However, now it is possible to work entirely through Django.

```
from django.http import JsonResponse    # replaces "import json"
from django.http import HttpResponse

def return_json(request):
    response_data = {}
    response_data['result'] = 'error'
    response_data['message'] = 'Some error message'
    return JsonResponse(response_data)  # abstraction to return json
```

Resources:

- Django - JsonResponse Objects

Procedure

1. Within a terminal space, move into your home directory.

   ```
   student@bchd:~$ cd
   ```

2. Make sure to install the virtual environment package.

   ```
   student@bchd:~$ sudo apt install virtualenv -y
   ```

3. Create the virtual env and then activate it.

   ```
   student@bchd:~$ python3 -m venv jsondja
   ```

4. Start your virtual environment.

   ```
   student@bchd:~$ source ./jsondja/bin/activate
   ```

5. Install Django within the virtual environment.

   ```
   (jsondja) student@bchd:~$ pip install Django
   ```

6. The next command creates a django project called `jsonreturn`. It automatically creates some directories and files to check out.

   ```
   (jsondja) student@bchd:~$ django-admin startproject jsonreturn
   ```

7. Now move into `~/jsonreturn/`

   ```
   (jsondja) student@bchd:~$ cd ~/jsonreturn/
   ```

8. Install `tree`. This has nothing to do with Django, it just lets us see directories from the CLI more easily.

   ```
   (jsondja) student@bchd:~/jsonreturn$ sudo apt install tree -y
   ```

9. Run `tree` to show the directory layout.

   ```
   (jsondja) student@bchd:~/jsonreturn$ tree
   ```

10. Notice that we have two folders named `jsonreturn`. The outer is the project name. The inner one is our main application. *We can have multiple applications within the project.*

11. Create a **new** `~/jsonreturn/jsonreturn/views.py` for the project.

    ```
    (jsondja) student@bchd:~/jsonreturn$ vim ~/jsonreturn/jsonreturn/views.py
    ```

12. Inside the `views.py` copy and paste the following:

```
#!/usr/bin/python3

from django.http import JsonResponse    # replaces "import json"
from django.http import HttpResponse

def schools(request):
    response_data = {}
    response_data['xmen'] = 'school for gifted youngsters'
    response_data['wizards'] = 'hogwarts school'
    response_data['vampires'] = 'forks high school'
    return JsonResponse(response_data)  # abstraction to return json
```

13. Save and exit with `:wq`

14. Take a look at the current copy of `~/jsonreturn/jsonreturn/urls.py`. This was created when we made our project. It gives access to the `/admin/` path, which we're not exploring right now.

```
(jsondja) student@bchd:~/jsonreturn$ cat ~/jsonreturn/jsonreturn/urls.py
```

15. Get rid of the default copy that was created for us.

```
(jsondja) student@bchd:~/jsonreturn$ rm ~/jsonreturn/jsonreturn/urls.py
```

16. Now open a new file called `urls.py`. Copy and paste the following into the script.

```
(jsondja) student@bchd:~/jsonreturn$ vim ~/jsonreturn/jsonreturn/urls.py
```

17. Create the following:

```
#!/usr/bin/python3

# imports from Django
from django.contrib import admin
from django.urls import path
from .import views

urlpatterns = [
    path('schools/', views.schools),
]
```

18. Save and exit with `:wq`

19. Establish a shell variable for the host we expect to send as a HTTP_HOST header to our Django server.

```
(jsondja) student@bchd:~/jsonreturn$ AUX1=aux1-`hostname --domain`.live.alta3.com
```

20. Change the settings.py file to allow for your aux1 HTTP_HOST header.

```
(jsondja) student@bchd:~/jsonreturn$ sed -i "s/ALLOWED_HOSTS = \[\]/ALLOWED_HOSTS = \['$AUX1', '127.0.0.1'\]/g" jsonreturn/settings.py
```

21. Allow CSRF Tokens to trust your host as well.

```
(jsondja) student@bchd:~/jsonreturn$ echo "CSRF_TRUSTED_ORIGINS = ['https://$AUX1', 'http://127.0.0.1']" >> jsonreturn/settings.py
```

22. Start your server on port 2224.

```
(jsondja) student@bchd:~/jsonreturn$ python3 manage.py runserver 0:2224
```

23. Split the screen with **CTRL** + **b** and then **SHIFT** + **"**

24. Curl the endpoint `/schools/` in verbose mode to see all of the headers. Django should have inserted a `content:` header. There should also be a message body with the JSON we created.

```
student@bchd:~$ curl http://127.0.0.1:2224/schools/ -v
```

25. If you want to scroll, press **CTRL** + **b** and then **[**. Use the arrow keys to scroll. Press `q` to exit the screen scroll.

26. Close the second panel with `exit`

```
student@bchd:~$ exit
```

27. Stop the Django test server with **CTRL** + **C**

28. Edit `~/jsonreturn/jsonreturn/views.py` for the project.

```
(jsondja) student@bchd:~/jsonreturn$ vim ~/jsonreturn/jsonreturn/views.py
```

29. Add the second function to your script. When you finish, your code should look like the following:

```
#!/usr/bin/python3

from django.http import JsonResponse    # replaces "import json"
from django.http import HttpResponse

def schools(request):
    response_data = {}
    response_data['xmen'] = 'school for gifted youngsters'
    response_data['wizards'] = 'hogwarts school'
    response_data['vampires'] = 'forks high school'
    return JsonResponse(response_data)  # abstraction to return json

# /?universe=marvel /?universe=dc
def readparams(request):

    response_data = {}

    universe = request.GET.get('universe', 'no universe found')
    response_data['universe'] = universe

    if universe == 'marvel':
        response_data['storm'] = 'control weather'
        response_data['cyclops'] = 'energy weapon (eye)'
        response_data['gambit'] = 'potential to kinetic energy'
    elif universe == 'dc':
        response_data['batman'] = 'expensive gadgets'
        response_data['superman'] = 'super everything'
        response_data['gambit'] = 'potential to kinetic energy'

    return JsonResponse(response_data)  # abstraction to return json
```

30. Save and exit with `:wq`

31. Now open a new file called `~/jsonreturn/jsonreturn/urls.py`. Copy and paste the following into the script.

```
(jsondja) student@bchd:~/jsonreturn$ vim ~/jsonreturn/jsonreturn/urls.py
```

32. Edit the `urlpattern` to include the entry for `comics/`:

```
#!/usr/bin/python3

# imports from Django
from django.contrib import admin
from django.urls import path
from .import views

urlpatterns = [
    path('schools/', views.schools),
    path('comics/', views.readparams),  #  comics/?universe=marvel
]
```

33. Save and exit with `:wq`

34. Start your server on port 2224.

```
(jsondja) student@bchd:~/jsonreturn$ python3 manage.py runserver 0:2224
```

35. Split the screen with **CTRL** + **b** and then **SHIFT** + **"**

36. Curl the endpoint `/comics/` in verbose mode. The 200 response should have have a fairly boring JSON response.

```
student@bchd:~$ curl http://127.0.0.1:2224/comics/ -v
```

37. If you want to scroll, press **CTRL** + **b** and then **[**. Use the arrow keys to scroll. Press `q` to exit the screen scroll.

38. Request information about the Marvel universe.

```
student@bchd:~$ curl http://127.0.0.1:2224/comics/?universe=marvel -v
```

39. If you want to scroll, press **CTRL** + **b** and then **[**. Use the arrow keys to scroll. Press `q` to exit the screen scroll.

40. Request information about the DC universe.

```
student@bchd:~$ curl http://127.0.0.1:2224/comics/?universe=dc -v
```

41. If you want to scroll, press **CTRL** + **b** and then **[**. Use the arrow keys to scroll. Press `q` to exit the screen scroll.

42. Close the second panel with `exit`

```
student@bchd:~$ exit
```

43. Stop the Django test server with **CTRL** + **C**

44. **CHALLENGE 01** - Create your own view that returns JSON. You can hard code any data set you'd like to return as JSON data. If you have knowledge of the `requests` library, you can use it to make your own API lookup.

45. Disconnect from the virtual environment.

```
(jsondja) student@bchd:~/jsonreturn$ deactivate
```

46. Return to the home directory.

```
(jsondja) student@bchd:~/jsonreturn$ cd
```

47. If you are saving your code via an SCM, ensure your code is within the `~/mycode` folder and then run the following command:

    - `cd`
    - `mv ~/jsonreturn/ ~/mycode/`
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "returning JSON with django"`
    - `git push origin`

53. Making requests with Django

Lab Objective

The objective of this lab is learn to make HTTP requests with Django. Consider the following:

```
# views.py

# python3 -m pip install requests
import requests
# python3 -m pip install Django
from django.http import JsonResponse

# In production, this should be set as an environment variable
API_KEY = "ABC123"

def enterprise_list(request):
    res = requests.get(f"https://www.example.com/api/json/v1/{ API_KEY }/search/s=blueprints")
    return JsonResponse(res.json())
```

The above view depends on the 3rd party library, `requests`. Therefore, to use this solution you need to run `python3 -m pip install requests`

Resources:

- Python - requests
- Django - JsonResponse Objects

Procedure

1. Within a terminal space, move into your home directory.

   ```
   student@bchd:~$ cd
   ```

2. Make sure to install the virtual environment package.

   ```
   student@bchd:~$ sudo apt install virtualenv -y
   ```

3. Create the virtual env and then activate it.

   ```
   student@bchd:~$ python3 -m venv djrequest
   ```

4. Start your virtual environment.

   ```
   student@bchd:~$ source ./djrequest/bin/activate
   ```

5. Install Django within the virtual environment.

   ```
   (djrequest) student@bchd:~$ pip install Django
   ```

6. The next command creates a django project called `djreq`. It automatically creates some directories and files to check out.

   ```
   (djrequest) student@bchd:~$ django-admin startproject djreq
   ```

7. Now move into `~/djreq/`

   ```
   (djrequest) student@bchd:~$ cd ~/djreq/
   ```

8. Install `tree`. This has nothing to do with Django, it just lets us see directories from the CLI more easily.

   ```
   (djrequest) student@bchd:~/djreq$ sudo apt install tree -y
   ```

9. Run `tree` to show the directory layout.

   ```
   (djrequest) student@bchd:~/djreq$ tree
   ```

10. Notice that we have two folders named `djreq`. The outer is the project name. The inner one is our main application. *We can have multiple applications within the project.*

11. Create a **new** `~/djreq/djreq/views.py` for the project.

    ```
    (djrequest) student@bchd:~/djreq$ vim ~/djreq/djreq/views.py
    ```

12. Inside the `views.py` copy and paste the following:

```
#!/usr/bin/python3

# python3 -m pip install requests
import requests

# python3 -m pip install Django
from django.http import JsonResponse    # replaces "import json"

# API to lookup - Django will proxy the request for us
API = "http://api.open-notify.org/astros.json"

def astro(request):
    res = requests.get(API)
    return JsonResponse(res.json())  # abstraction to return json
```

13. Save and exit with `:wq`

14. In order to use this solution, we need to install requests.

```
(djrequest) student@bchd:~/djreq$ python3 -m pip install requests
```

15. Take a look at the current copy of `~/djreq/djreq/urls.py`. This was created when we made our project. It gives access to the `/admin/` path, which we're not exploring right now.

```
(djrequest) student@bchd:~/djreq$ cat ~/djreq/djreq/urls.py
```

16. Get rid of the default copy that was created for us.

```
(djrequest) student@bchd:~/djreq$ rm ~/djreq/djreq/urls.py
```

17. Now open a new file called `urls.py`. Copy and paste the following into the script.

```
(djrequest) student@bchd:~/djreq$ vim ~/djreq/djreq/urls.py
```

18. Create the following:

```
#!/usr/bin/python3

# imports from Django
from django.contrib import admin
from django.urls import path
from .import views

urlpatterns = [
    path('astro/', views.astro),
]
```

19. Save and exit with `:wq`

20. Establish a shell variable for the host we expect to send as a HTTP_HOST header to our Django server.

```
(djrequest) student@bchd:~/djreq$ AUX1=aux1-`hostname --domain`.live.alta3.com
```

21. Change the settings.py file to allow for your aux1 HTTP_HOST header.

```
(djrequest) student@bchd:~/djreq$ sed -i "s/ALLOWED_HOSTS = \[\]/ALLOWED_HOSTS = \['$AUX1', '127.0.0.1'\]/g" djreq/settings.py
```

22. Allow CSRF Tokens to trust your host as well.

```
(djrequest) student@bchd:~/djreq$ echo "CSRF_TRUSTED_ORIGINS = ['https://$AUX1', 'http://127.0.0.1']" >> djreq/settings.py
```

23. Start your server on port 2224.

```
(djrequest) student@bchd:~/djreq$ python3 manage.py runserver 0:2224
```

24. Split the screen with **CTRL** + **b** and then **SHIFT** + **"**

25. Curl the endpoint `/astro/`. It should return the information available from `http://api.open-notify.org/astros.json`

```
student@bchd:~$ curl http://127.0.0.1:2224/astro/
```

26. Close the second panel with `exit`

```
student@bchd:~$ exit
```

27. Stop the Django test server with **CTRL** + **C**

28. Edit `~/djreq/djreq/views.py` for the project.

```
(djrequest) student@bchd:~/djreq$ vim ~/djreq/djreq/views.py
```

29. Add the second function to your script. When you finish, your code should look like the following:

```
#!/usr/bin/python3

# python3 -m pip install requests
import requests

# python3 -m pip install Django
from django.http import JsonResponse    # replaces "import json"

# API to lookup - Django will proxy the request for us
API = "http://api.open-notify.org/astros.json"

# https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY
APINASA = "https://api.nasa.gov/planetary/apod?api_key="

# Your NASA API key goes here
# in production this should be set as an environmental variable
APIKEY = "DEMO_KEY"

def astro(request):
    res = requests.get(API)
    return JsonResponse(res.json())  # abstraction to return json

def nasa(request):
    res = requests.get(f"{APINASA}{APIKEY}")
    return JsonResponse(res.json()) # abstraction to return json
```

30. Save and exit with `:wq`

31. Now open a new file called `~/djreq/djreq/urls.py`. Copy and paste the following into the script.

```
(djrequest) student@bchd:~/djreq$ vim ~/djreq/djreq/urls.py
```

32. Edit the `urlpattern` to include the entry for `nasa/`:

```
#!/usr/bin/python3

# imports from Django
from django.contrib import admin
from django.urls import path
from .import views

urlpatterns = [
    path('astro/', views.astro),    # access by proxy api.open-notify.org/astros.json
    path('nasa/', views.nasa),      # access by proxy api.nasa.gov
]
```

33. Save and exit with `:wq`

34. Start your server on port 2224.

```
(djrequest) student@bchd:~/djreq$ python3 manage.py runserver 0:2224
```

35. Split the screen with **CTRL** + **b** and then **SHIFT** + **"**

36. Curl the endpoint `/nasa/`

```
student@bchd:~$ curl http://127.0.0.1:2224/nasa/
```

37. Close the second panel with `exit`

```
student@bchd:~$ exit
```

38. Stop the Django test server with **CTRL** + **C**

39. **CHALLENGE 01** - Create your own view and urlpattern that performs and API request. It can be to anywhere you'd like, but here are some suggestions:

    - Try another API endpoint at api.nasa.gov
    - Have Django catch some Pokemon from https://pokeapi.co/ (no API key required)
    - Return some SpaceX data from the service described at https://github.com/r-spacex/SpaceX-API. Many of the endpoints do not require authentication, for example: https://api.spacexdata.com/v4/launches

40. **CHALLENGE 02** - Update the view `nasa` so that the API key may be passed in by the GET request, for example, `curl http://127.0.0.1:2224/nasa/?apikey=1a2b3c4d5e` should use the key `1a2b3c4d5e` as an API key to the api.nasa.gov service.

41. Disconnect from the virtual environment.

```
(djrequest) student@bchd:~/djreq$ deactivate
```

42. Return to the home directory.

```
(djrequest) student@bchd:~/djreq$ cd
```

43. If you are saving your code via an SCM, ensure your code is within the `~/mycode` folder and then run the following command:
    - `cd`
    - `mv ~/djreq/ ~/mycode/`
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "making requests with Django"`
    - `git push origin`

54. Django App Design - To-Do app

The objective of this lab is to explore building applications within Django. The application we'll build is a to-do application. Within this application, we'll use a library named `django-crispy-forms`. This gives us access to the `crispy` filter and tag within templates. We'll use this library to assist in building our interactive forms.

Resources:

- Django crispy forms
- Django Documentation - Creating forms from models

Procedure

1. Within a terminal space, move into your home directory.

```
student@bchd:~$ cd
```

2. Make a new directory. *It is okay if this directory still exists*

```
student@bchd:~$ mkdir mycode
```

3. Move into `~/mycode`

```
student@bchd:~$ cd ~/mycode
```

4. Create a virtual environment for your todo app.

```
student@bchd:~/mycode$ python3 -m venv todo_env
```

5. Source your virtual environment.

```
student@bchd:~/mycode$ source todo_env/bin/activate
```

6. Make sure necessary tools are installed.

```
(todo_env) student@bchd:~/mycode$ python3 -m pip install virtualenv Django django-crispy-forms
```

7. Start the project.

```
(todo_env) student@bchd:~/mycode$ django-admin startproject todo_site
```

8. Move into your project directory.

```
(todo_env) student@bchd:~/mycode$ cd todo_site
```

9. Make the todo app.

```
(todo_env) student@bchd:~/mycode/todo_site$ python3 manage.py startapp todo
```

10. Install tree *this is a operating system tool, it has nothing to do with Django or Python*

```
(todo_env) student@bchd:~/mycode/todo_site$ sudo apt install tree -y
```

11. Run tree.

```
(todo_env) student@bchd:~/mycode/todo_site$ tree
```

12. Move into the `todo/` app dir.

```
(todo_env) student@bchd:~/mycode/todo_site$ cd todo
```

13. Make a directory for your templates to live in.

```
(todo_env) student@bchd:~/mycode/todo_site/todo$ mkdir ~/mycode/todo_site/todo/templates/
```

14. Create a new `~/mycode/todo_site/todo/templates/index.html` file.

```
(todo_env) student@bchd:~/mycode/todo_site$ vim ~/mycode/todo_site/todo/templates/index.html
```

15. Create the following template, it contains both HTML and CSS data that will render our ToDo page. Notable is the use of the `crispy_forms` package that helps a developer write as little HTML as possible. Also note the use of variables like, `{{title}}` and logic conditions like `{% if messages %}`. This usage is similar to Jinja.

```
{% load crispy_forms_tags %}
<!DOCTYPE html>
<html lang="en" dir="ltr">

<head>

  <meta charset="utf-8">
  <title>{{title}}</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
  <!--style-->
  <style>
  .card {

   box-shadow: 0 4px 8px 0 rgba(0,0,0,0.5),
               0 6px 20px 0 rgba(0,0,0,0.39);
   background: lightpink;
   margin-bottom : 5%;
   border-radius: 25px;
   padding : 2%;
   overflow: auto;
   resize: both;
   text-overflow: ellipsis;
  }
  .card:hover{
    background: lightblue;
  }

  .submit_form{

    text-align: center;
    padding: 3%;
    background: pink;
    border-radius: 25px;
    box-shadow: 0 4px 8px 0 rgba(0,0,0,0.4),
                0 6px 20px 0 rgba(0,0,0,0.36);
  }
  </style>

</head>

<body  class="container-fluid">

  {% if messages %}
  {% for message in messages %}
  <div class="alert alert-info">
    <strong>{{message}}</strong>
  </div>
  {% endfor %}
  {% endif %}

  <center class="row">
    <h1><i>TODO LIST</i></h1>
    <hr />
  </center>

  <div class="row">

    <div class="col-md-8">

      {% for i in list %}
      <div class="card">
        <center><b>{{i.title}}</b></center>
        <hr/>
        {{i.date}}
        <hr/>
        {{i.details}}
        <br />
        <br />
        <form action="/del/{{i.id}}" method="POST" style=" padding-right: 4%; padding-bottom: 3%;">
          {% csrf_token %}
          <button value="remove" type="submit"  class="btn btn-primary" style="float: right;"><span class="glyphicon glyphicon-trash"></span>  
remove</button>
        </form>
      </div>
      {% endfor%}
    </div>
    <div class="col-md-1"> </div>
    <div class="col-md-3" >
      <div  class="submit_form">
      <form  method="POST">
        {% csrf_token %}
```

```
          {{forms|crispy}}
          <center>
          <input type="submit" class="btn btn-default" value="submit" />
        </center>
        </form>
      </div>
    </div>
  </div>
</body>

</html>
```

16. Save and exit with `:wq`

17. Move into the `~/mycode/todo_site/` directory

```
(todo_env) student@bchd:~/mycode/todo_site/todo$ cd ~/mycode/todo_site/
```

18. Edit your settings file.

```
(todo_env) student@bchd:~/mycode/todo_site$ vim ~/mycode/todo_site/todo_site/settings.py
```

19. Add the two lines at the bottom of the `INSTALLED_APPS`, `todo` and `crispy_forms`. The first is the name of our app. The second is the plugin we're using to help create our templates. Only the apps listed here have access to things like models (i.e. databases), static files, tests, and management commands.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'todo',
    'crispy_forms',
]
```

20. Press **Esc** then save and exit with `:wq`

21. Establish a shell variable for the host we expect to send as a HTTP_HOST header to our Django server.

```
(todo_env) student@bchd:~/mycode/todo_site$ AUX1=aux1-`hostname --domain`.live.alta3.com
```

22. Change the `settings.py` file to allow for your aux1 HTTP_HOST header.

```
(todo_env) student@bchd:~/mycode/todo_site$ sed -i "s/ALLOWED_HOSTS = \[\]/ALLOWED_HOSTS = \['$AUX1','127.0.0.1'\]/g" todo_site/settings.py
```

23. Allow Django to trust your host using CSRF.

```
(todo_env) student@bchd:~/mycode/todo_site$ echo "CSRF_TRUSTED_ORIGINS = ['https://$AUX1','http://127.0.0.1']" >> todo_site/settings.py
```

24. Make a URLs python file for our todo_site.

```
(todo_env) student@bchd:~/mycode/todo_site$ vim todo_site/urls.py
```

25. Update `todo_site/urls.py` to look like the following:

```
from django.contrib import admin
from django.urls import path
from todo import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name='todo'),
    # give id no. item_id name or item_id=i.id
    # pass item_id as primary key to remove that todo item with given id
    path('del/<str:item_id>', views.remove, name='del'),
]
```

26. Press **Esc** then save and exit with `:wq`

27. Edit your models. A *model* is a pythonic description of what should appear in the database. By default, Django uses SQLite, but it also officially supports PostgreSQL, MariaDB, MySQL, and Oracle are all supported.

```
(todo_env) student@bchd:~/mycode/todo_site$ vim todo/models.py
```

28. Create the following model:

```
from django.db import models
from django.utils import timezone

# the name of our table is "todo"
class Todo(models.Model):
        title=models.CharField(max_length=100)  # create column "title"
        details=models.TextField()              # create column "details"
        date=models.DateTimeField(default=timezone.now)    # create column "date"

        # no need to create an entry for primary_id
        # django will take care of this

        def __str__(self):
                return self.title
```

29. Press **Esc** then save and exit with `:wq`

30. Edit your views.py.

```
(todo_env) student@bchd:~/mycode/todo_site$ vim ~/mycode/todo_site/todo/views.py
```

31. Create the following:

```
from django.shortcuts import render, redirect
from django.contrib import messages

## import todo form and models

from .forms import TodoForm
from .models import Todo

###############################################

def index(request):

        item_list = Todo.objects.order_by("-date")
        if request.method == "POST":
                form = TodoForm(request.POST)
                if form.is_valid():
                        form.save()
                        return redirect('todo')
        form = TodoForm()

        page = {
                        "forms" : form,
                        "list" : item_list,
                        "title" : "TODO LIST",
                }
        return render(request, 'index.html', page)



### function to remove item, it receive todo item_id as primary key from url ##
def remove(request, item_id):
        item = Todo.objects.get(id=item_id)
        item.delete()
        messages.info(request, "item removed !!!")
        return redirect('todo')
```

32. Press **Esc** then save and exit with `:wq`

33. Edit your forms.py.

```
(todo_env) student@bchd:~/mycode/todo_site$ vim ~/mycode/todo_site/todo/forms.py
```

34. If you're building a database-driven app, chances are you'll have forms that map closely to Django models. For instance, you might have a Todo model, and you want to create a form that lets people submit Todo tasks. In this case, it would be redundant to define the field types in your form, because you've already defined the fields in your model. For this reason, Django provides a helper class that lets you create a Form class from a Django model called `ModelForm`.

```
from django import forms
from .models import Todo

class TodoForm(forms.ModelForm):
        class Meta:
                model = Todo
                fields="__all__"
```

35. Press **Esc** then save and exit with `:wq`

36. Register models to admin. This allows an admin user to interact with the model (database) through a special admin page (`admin/`).

```
student@bchd:~/mycode/todo_site$ vim ~/mycode/todo_site/todo/admin.py
```

37. Create the following.

```
from django.contrib import admin
from .models import Todo

# the admin site has access to the Todo model
admin.site.register(Todo)
```

38. Press **Esc** then save and exit with `:wq`

39. Make migrations.

```
(todo_env) student@bchd:~/mycode/todo_site$ python3 manage.py makemigrations
```

40. Migrate your changes to the database.

```
(todo_env) student@bchd:~/mycode/todo_site$ python3 manage.py migrate
```

41. Start the server on port `2224`

```
(todo_env) student@bchd:~/mycode/todo_site$ python3 manage.py runserver 0:2224
```

42. Choose the `aux1` pop-out from the drop down menu in the upper right corner (press the arrow beside `aux1` to pop-out to a new tab).

43. Your **TODO List** should appear. It is fairly straight forward how to use it. Try it out!

44. Return to the terminal.

45. Stop the server with **CTRL** + **C**

46. **CHALLENGE 01** - Run the command `python manage.py createsuperuser` from within your project directory to create an admin user. After doing so, start your project and navigate to `admin/` within aux1. Login with the user. You should see the Todo model. If you click on it, you may interact with the database. Try loading some new tasks, and then making sure the show up when you visit the todo application.

47. Deactivate the virtual environment.

```
(todo_env) student@bchd:~/mycode/todo_site$ deactivate
```

48. If you are saving your code via an SCM, ensure your code is within the `~/mycode` folder and then run the following command:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "todo list"`
    - `git push origin`

55. Swagger

Appropriate API architecting is essential to designing an API that will be easily usable, quickly adopted, and resilient against future changes.

API Design Best Practice

Most modern APIs are designed to use HTTP as the protocol, so it is worth going over the most common HTTP methods that we can use.

CRUD	Method	Description
Create	POST	Create new resources
Read	GET	Retrieve a representation of a resource
Update	PUT	Update existing resources
Delete	DELETE	Delete existing resources

Developers like to use these HTTP Methods as *verbs* which make things happen on the resource paths. So best practice for designing a URL is to have it include *nouns*, such as `/pets` or `/users`. Then a developer can effectively say `GET /users/<username>` or `DELETE /users/<username>`. This allows the developer to use the HTTP Method as the directive on a specific resource and be very clear in what they are trying to do.

Another important aspect of a well designed API is to always have examples. That way a developer can very quickly understand what an appropriate HTTP request should look like, and also what the expected response would be.

Lab Objective

The objective of this lab is to learn to document APIs with Swagger. Swagger is a standard tool that is built with YAML (or JSON). If you've never studied YAML, then be sure to check out the YAML spec here: https://yaml.org/spec/1.2/spec.html

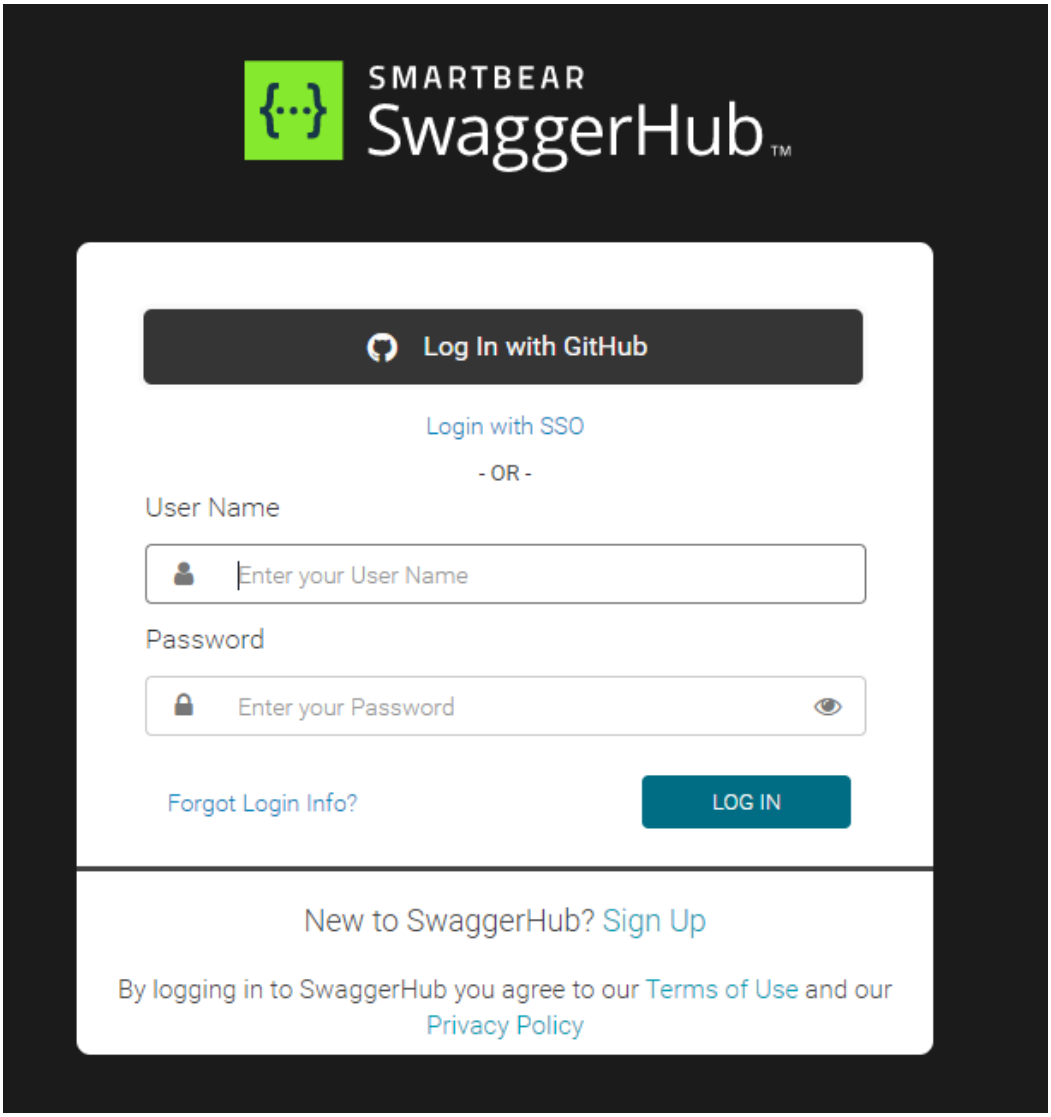
Swagger (v2) and OpenAPI (v3) are public tools free to use and licensed under the Apache 2.0 License. The source code is available here: https://github.com/swagger-api

Swagger also offers **Swagger Hub**, which you can use to explore Swagger, and document APIs. This tool also allows you to sign up for collaboration across organizations, which is a pay-to-use service.

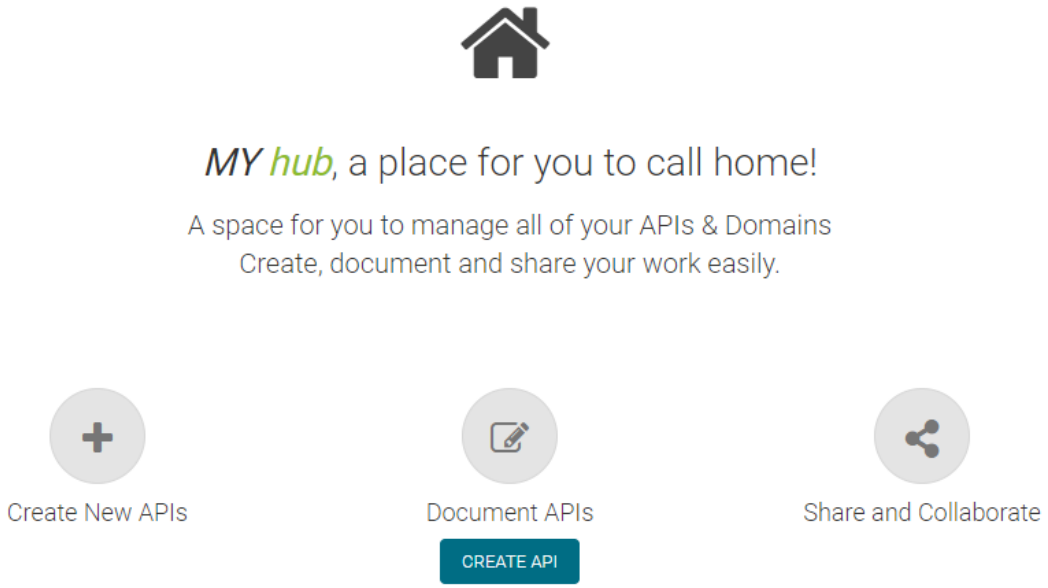
If you want to read a detailed summary of Swagger Hub, check out the following: https://app.swaggerhub.com/help/

Procedure

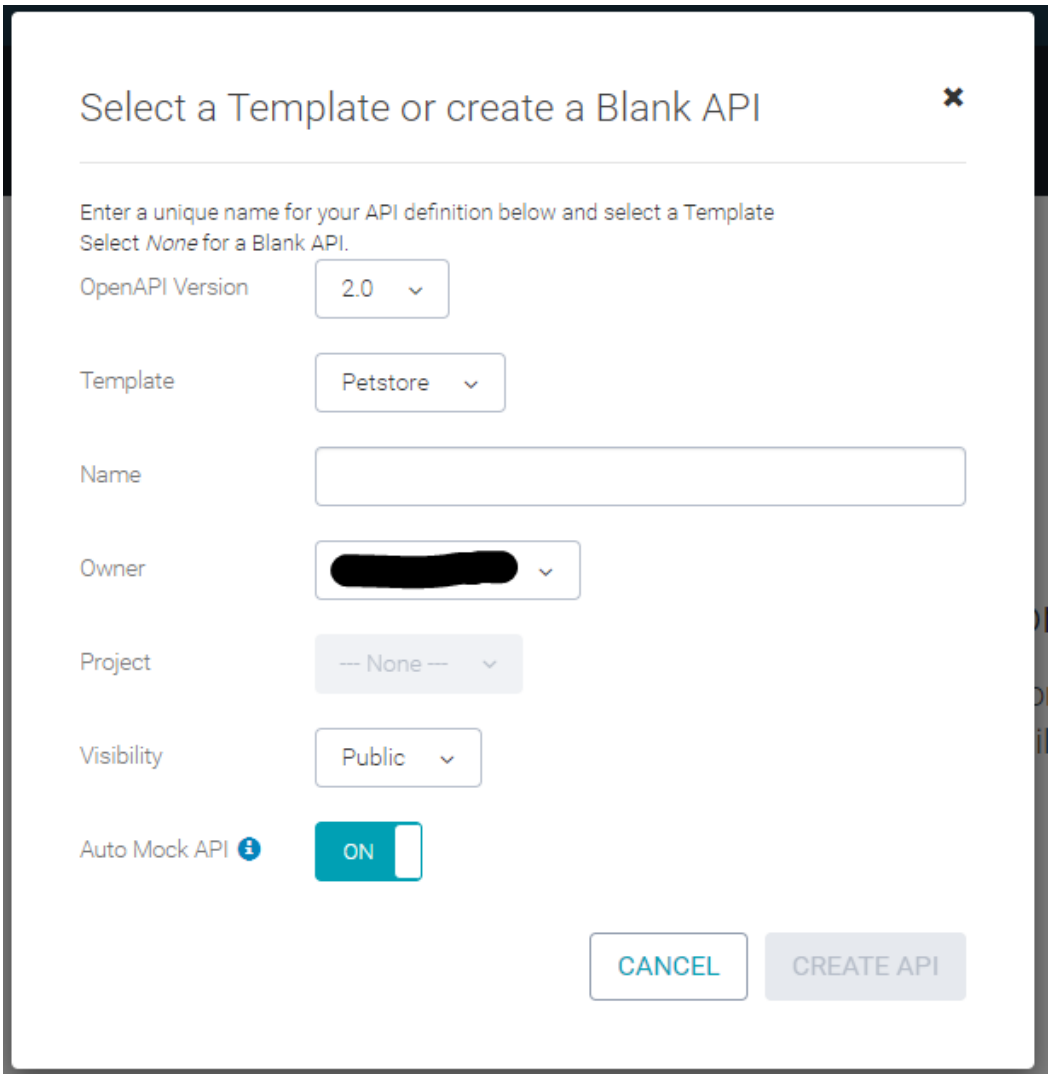
1. Want a FREE personal swagger account? In your browser, navigate to https://swagger.io/. Then click on the top right where it says "Log In". This will give you the option to Log in with GitHub, SSO, or SwaggerHub sign-in. We suggest you use your GitHub login.



2. Next you should be greeted by a page like this:



3. Click on the **CREATE API** button. You will then see a screen like this:



Keep all of the settings as seen here, and give it your own name. Then click on the **CREATE API** button.

4. After a few seconds, you should see a new page pop up. The center will have an area to edit the swagger.yaml file which has the design of the API laid out for you. Let's start to dissect it a bit.

Lines 1 - 17: These lines describe the version and information of Swagger itself.

```
swagger: '2.0'
info:
  description: |
    This is a sample Petstore server.  You can find
    out more about Swagger at
    [http://swagger.io](http://swagger.io) or on
    [irc.freenode.net, #swagger](http://swagger.io/irc/).
  version: 1.0.0
  title: Swagger Petstore
  termsOfService: http://swagger.io/terms/
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
# host: petstore.swagger.io
# basePath: /v2
```

Lines 18 - 32: These lines describe tags that are being applied inside of your API. Think of these as labels that Swagger uses to group pieces of your API together. Read more about them here: https://swagger.io/docs/specification/grouping-operations-with-tags/

```
tags:
- name: pet
  description: Everything about your Pets
  externalDocs:
    description: Find out more
    url: http://swagger.io
- name: store
  description: Access to Petstore orders
- name: user
  description: Operations about user
  externalDocs:
    description: Find out more about our store
    url: http://swagger.io
# schemes:
# - http
```

Lines 33 - 561: These lines describe your **paths**. For obvious reasons, they are not all shown here. The `/pet` path has two optional HTTP Request methods: **post** and **put**. They each are **tag**ged, have a **summary, operationID, consumes, produces, parameters, responses, and security**. Each of these will be discussed in depth further into this lab, but for right now, just read through this example to try to figure out for yourself what each one of these may be doing.

```
paths:
  /pet:
    post:
      tags:
      - pet
      summary: Add a new pet to the store
      operationId: addPet
      consumes:
      - application/json
      - application/xml
      produces:
      - application/json
      - application/xml
      parameters:
      - in: body
        name: body
        description: Pet object that needs to be added to the store
        required: true
        schema:
          $ref: '#/definitions/Pet'
      responses:
        405:
          description: Invalid input
      security:
      - petstore_auth:
        - write:pets
        - read:pets
    put:
      tags:
      - pet
      summary: Update an existing pet
      operationId: updatePet
      consumes:
      - application/json
      - application/xml
      produces:
      - application/json
      - application/xml
      parameters:
      - in: body
        name: body
        description: Pet object that needs to be added to the store
        required: true
        schema:
          $ref: '#/definitions/Pet'
      responses:
        400:
          description: Invalid ID supplied
        404:
          description: Pet not found
        405:
          description: Validation exception
      security:
      - petstore_auth:
        - write:pets
        - read:pets
      ...
```

Lines 562 - 573: This is where you define all authentication types that will be supported by the API. You then reference these using the **security** term in your paths to apply one or more specific authentication types to the individual paths. Read more about it here: https://swagger.io/docs/specification/2-0/authentication/

```
securityDefinitions:
  petstore_auth:
    type: oauth2
    authorizationUrl: http://petstore.swagger.io/oauth/dialog
    flow: implicit
    scopes:
      write:pets: modify pets in your account
      read:pets: read your pets
  api_key:
    type: apiKey
    name: api_key
    in: header
```

Lines 574 - 682: These definitions are where you define various objects that you will use throughout your API. Think of these like Python Classes, where these Objects each have attributes.

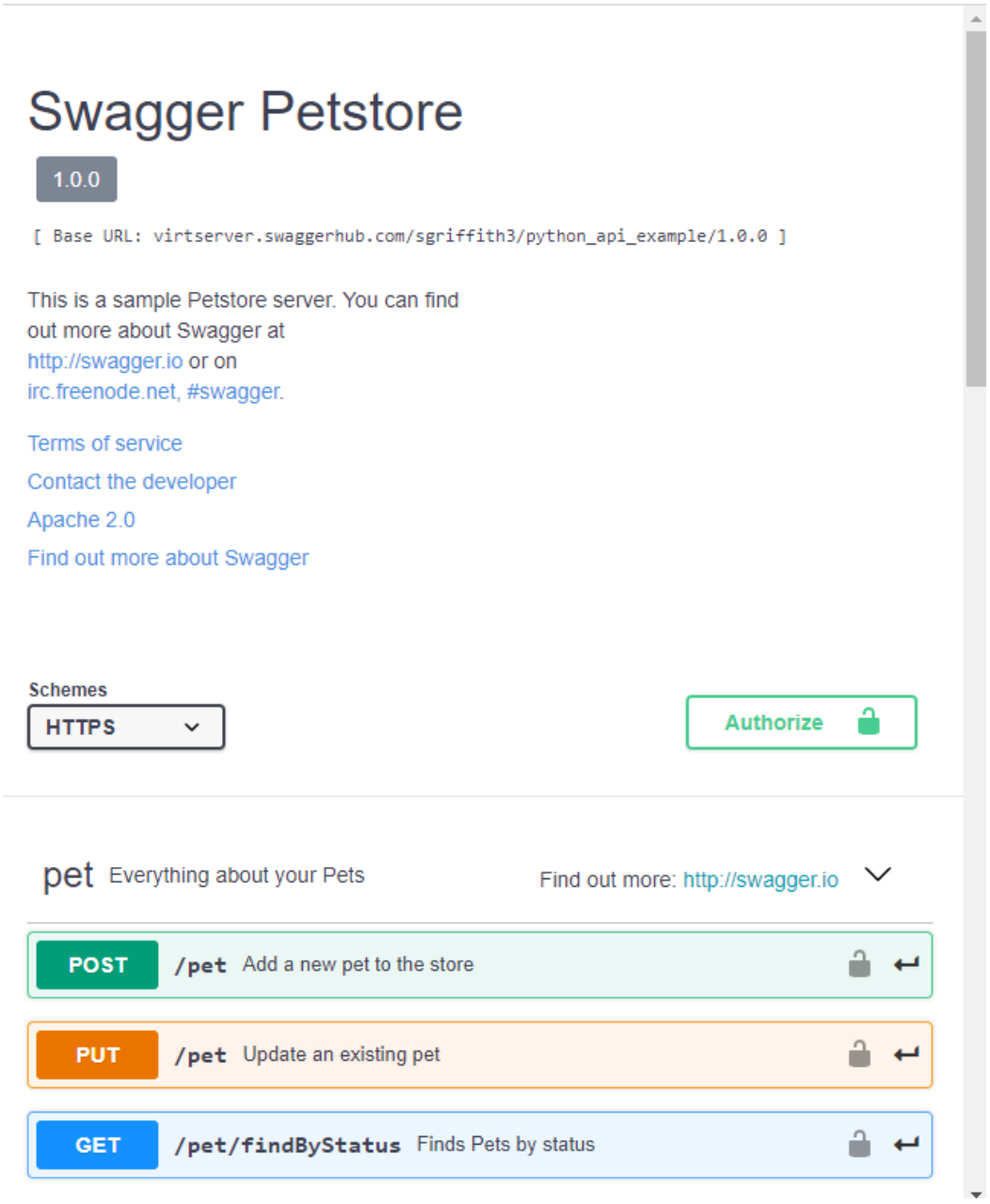
```
definitions:
  Pet:
    type: object
    required:
    - name
    - photoUrls
    properties:
      id:
        type: integer
        format: int64
      category:
        $ref: '#/definitions/Category'
      name:
        type: string
        example: doggie
      photoUrls:
        type: array
        xml:
          name: photoUrl
          wrapped: true
        items:
          type: string
      tags:
        type: array
        xml:
          name: tag
          wrapped: true
        items:
          $ref: '#/definitions/Tag'
      status:
        type: string
        description: pet status in the store
        enum:
        - available
        - pending
        - sold
    xml:
      name: Pet
```

Lines 693 - 701: These describe the **externalDocs**, which **host** this API is being served on, what the **basePath** is, as well as what **schemes** are available.

```
externalDocs:
  description: Find out more about Swagger
  url: http://swagger.io
# Added by API Auto Mocking Plugin
host: virtserver.swaggerhub.com
basePath: /sgriffith3/python_api_example/1.0.0
schemes:
 - https
 - http
```

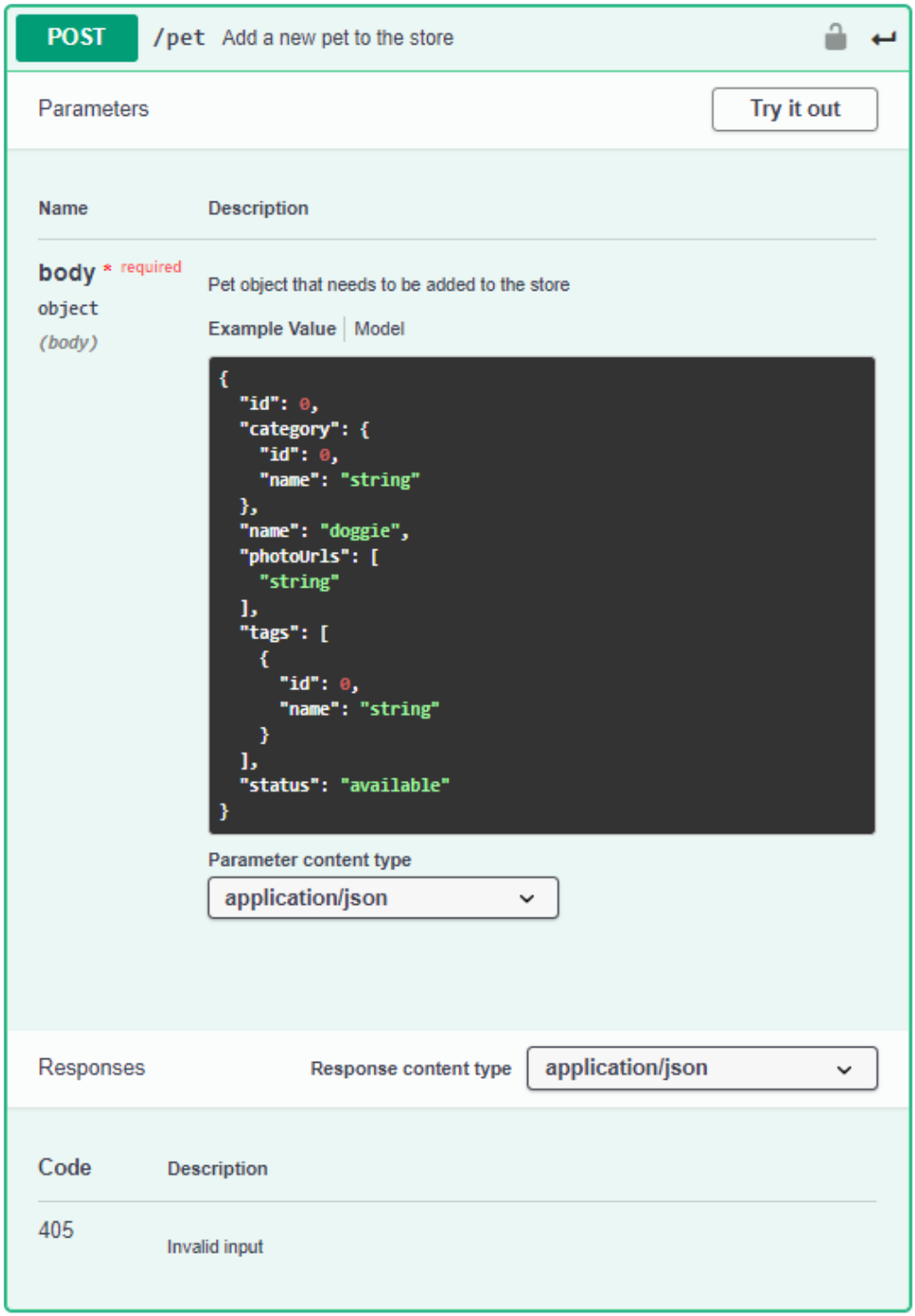
5. If you are looking for some more details about the basic structure of Swagger API definitions, check out the documentation: https://swagger.io/docs/specification/2-0/basic-structure/

6. Now let's take a look over at the right hand side of the screen.



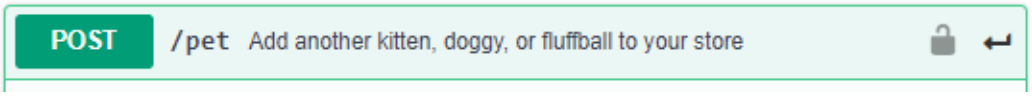
Notice how the path of **/pet** has two nicely color-coded options of either performing a **POST** or a **PUT** command. These get *generated from the YAML definition*!

7. Click on the top **POST /pet** area. When you do this, it will expand and you will be able to see the Parameters and Responses, as well as the example data. **NOW LOOK AT THE YAML DEFINITION (lines 34 - 59).**



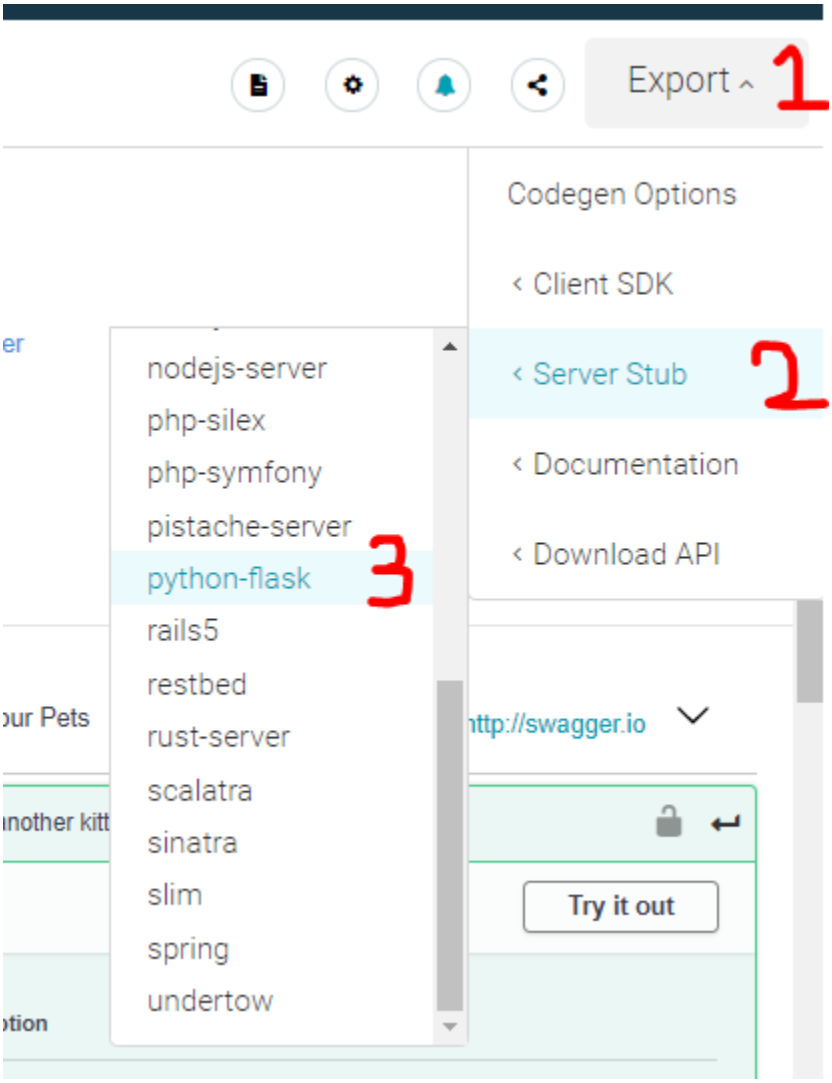
All of this information was generated from your YAML definition.

8. On line 38, change the **summary** value to "Add another kitten, doggy, or fluffball to your store". After you do this, you should see that the summary for the POST has gotten updated on the right hand side.



This proves that you can have the documentation get dynamically updated when you make a change to any portion of this API Definition!

9. Next we will make 3 clicks to have swagger *generate a python flask server* for you! At the top right, click **Export** -> **Server Stub** -> **python-flask**. See this picture for the flow of your clicks.



> This will download a python flask server for you. All you have to do now is read through the README.md file, and follow the steps therein to start your server.

10. Nothing is perfect, and unfortunately this includes their code as well. It is a little bit busted and requires a few tweaks. Thankfully we have a working version ready to go for you. So let's download the *working* code now.

```
student@bchd:~$ wget https://static.alta3.com/courses/python/swagger_flask_petstore.tar
```

11. Next up we will need to unpack this little tarball.

```
student@bchd:~$ tar -xvf swagger_flask_petstore.tar
```

12. Move into the swagger_flask_petstore directory.

```
student@bchd:~$ cd swagger_petstore_flask
```

13. Perform a pip install of all of our requirements.

```
student@bchd:~/swagger_petstore_flask$ python3 -m pip install -r requirements.txt
```

> If you have run the pandas lab, you may see an error with one of the requirements versions. As long as the one installed is newer than the requirements.txt asks for, it should be okay.

14. Now start up your server by calling on the module's name.

```
student@bchd:~/swagger_petstore_flask$ python3 -m swagger_server
```

> Note: On Feb 17, 2022 Markupsafe pushed release 2.1.0 that removed soft_unicode - Making it likely that this code failed for you. *This is an example of why it is important to explicitly declare the dependencies (and their versions) that your code needs!*

15. If you see an error from the previous step:

```
student@bchd:~/swagger_petstore_flask$ python3 -m pip uninstall markupsafe flask-limiter
```

Hit "y" when prompted

16. Then install the correct verson of Markupsafe:

```
student@bchd:~/swagger_petstore_flask$ python3 -m pip install markupsafe==2.0.1
```

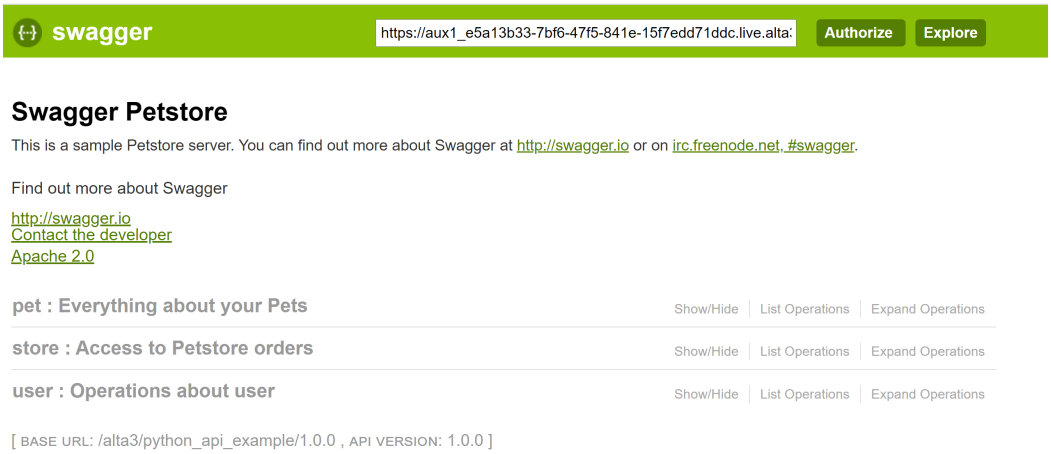
17. Now start up your server by calling on the module's name. It should work this time.

```
student@bchd:~/swagger_petstore_flask$ python3 -m swagger_server
```

18. Open up your **aux1** terminal and append the following to your URI.

```
/alta3/python_api_example/1.0.0/ui/
```

You should be greeted by the following screen



56. Glossary

- **Attribute** - Values associated with an individual object. Attributes are accessed using the 'dot syntax': a.x means fetch the x attribute from the 'a' object.
- **BDFL - Acronym for "Benevolent Dictator For Life"** - a.k.a. Guido van Rossum, Python's primary creator, figurehead and decision-maker.
- **EAFP** - Acronym for the saying it's "Easier to Ask for Forgiveness than Permission". This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the LBYL style that is common in many other languages such as C.
- **EIBTI** - Acronym for "Explicit Is Better Than Implicit", one of Python's design principles, included in the Zen of Python.
- **IDLE** - an Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment that ships with the standard distribution of Python. Good for beginners and those on a budget, it also serves as clear example code for those wanting to implement a moderately sophisticated, multi-platform GUI application.
- **LBYL** - Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized the presence of many if statements.
- **Python 3000** - older name for Python 3. An update to the language that allowed breaking changes to be made in order to add features and improve various functionality.
- **Zen of Python** - listing of Python design principles and philosophies that are helpful in understanding and using the language effectively. The listing can be found by typing "import this" at the interactive prompt.
- **slots** - A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory critical application.
- **byte code** - The internal representation of a Python program in the interpreter. The byte code is also cached in .pyc and .pyo files so that executing the same file is faster the second time (the step of compilation from source to byte code can be saved). This "intermediate language" is said to run on a "virtual machine" that calls the subroutines corresponding to each bytecode.
- **class** - A template for creating user-defined objects. Class definitions normally contain method definitions that operate on instances of the class.
- **coercion** - The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, int(3.15) converts the floating point number to the integer, 3, but in 3 + 4.5, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a TypeError. Coercion between two operands can be implicitly invoked with the coerce builtin function; thus, 3 + 4.5 is equivalent to operator.add(*coerce(3, 4.5)) and results in operator.add(3.0, 4.5) which is of course 7.5. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., float(3) + 4.5 rather than just 3 + 4.5.
- **complex number** - An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit, often written i in mathematics or j in engineering. Python has builtin support for complex numbers, which are written with this latter notation; the imaginary part is written with a j suffix, e.g., 3+1j. To get access to complex equivalents of the math module, use cmath. Use of complex numbers is a fairy advanced mathematical feature; if you're not aware of a need for complex numbers, it's almost certain you can safely ignore them.
- **conversion** - The invocation of a well-defined mechanism from of transforming an instance of one type of object to an instance of another; for example, int('3') will convert a string ('3') to an int (3).
- **decorator** - A function that modifies another function or method. Its return value is typically a callable object, possibly the original function, but most often another function that modifies the original function's behavior in some fashion.
- **descriptor** - Any object that defines the methods **get**(), **set**(), or **delete**(). When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, writing a.b looks up the object b in the class dictionary for a, but if b is a descriptor, the defined method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods,properties, class methods, static methods, and reference to super classes.
- **dictionary** - A built-in Python data type composed of arbitrary keys and values; sometimes called a "hash" or a "hash map" in other languages, although this is technically a misnomer (hashing is one way to implement an associative array but not the only way). The use of dict much resembles that for list, but the keys can be any object with a **hash** function, not just integers starting from zero. Examples: d = {'A':65, 'B':66}, d = dict([('A', 65), ('B', 66)]), d['C'] = 67
- **docstring** - A string that appears as the lexically first expression in a module, class definition or function/method definition is assigned as the **doc** attribute of the object where it is available to documentation tools or the help() builtin function.
- **duck typing** - From the "If it walks, talks, and looks like a duck, then it's a duck" principle. Python uses duck typing in that if an object of some user-defined type exhibits all of the expected interfaces of some type (say the string type), then the object can be treated as if it really were of that type.
- **dynamic typing** - A style of typing of variables where the type of objects to which variables are assigned can be changed merely by reassigning the variables. Python is dynamically typed. Thus, unlike as in a statically typed language such as C, a variable can first be assigned a string, then an integer, and later a list, just by making the appropriate assignment statements. This frees the programmer from managing many details, but does come at a performance cost.
- **function** - A block of code that is invoked by a "calling" program, best used to provide an autonomous service or calculation.
- **generator function** - A function that returns a generator iterator. Its definition looks like a normal function definition except that it uses the keyword yield. Generator functions often contain one or more for or while loops that yield elements. The function execution is stopped at the yield keyword (returning the result) and its resumed there when the next element is requested (e.g., by the builtin function next()). For details see PEP 0255 and PEP 0342.
- **generator** - The common name for a generator iterator. The type of iterator returned by a generator function or a generator expression.
- **global interpreter lock or GIL** - the lock used by Python threads to assure that only one thread can be run at a time. This simplifies Python by assuring that no two processes can access the same memory at the same time. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of some parallelism on multi-processor machines. Efforts have been made in the past to create a "free-threaded" interpreter (one which locks shared data at a much finer granularity), but performance suffered in the common single-processor case. See GlobalInterpreterLock.
- **greedy regular expressions** - Regular expressions which match the longest string possible. The *, + and ? operators are all greedy. Their counterparts *?, +? and ?? are all non-greedy (match the shortest string possible).
- **hash table** - An object that maps more-or-less arbitrary keys to values. Dictionaries are the most visible and widely used objects that exhibit this behavior.
- **hash** - A number used to correspond to objects, usually used for 'hashing' keys for storage in a hash table. Hashing in Python is done with the builtin hash function
- **hashable** - An object is hashable if it is immutable (ints, floats, tuples, strings, etc) or user-defined classes that define a **hash** method.
- **id** - id is a built-in function which returns a number identifying the object, referred to as the object's id. It will be unique during the lifetime of the object, but is very often reused after the object is deleted.
- **immutable** - An object with fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed such as with keys in a dictionary.

- **integer division** - Mathematical division discarding any remainder, for example 3 // 2 returns 1, in contrast to the 1.5 returned by float division. Also called "floor division". When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a float), the result will be coerced (see coercion) to a common type. For example, an integer divided by a float will result in a float value. Integer division can be carried out by using the '//' operator instead of the '/' operator (used for "true division").
- **interactive** - Python has an interactive interpreter which means that you can try out things and directly see its result. To use it, launch python3 with no arguments. A very powerful way to test out new ideas, inspect libraries (remember x.**doc** and help(x)) and improve programming skills.
- **interpreted** - Python is an interpreted language (like Perl), as opposed to a compiled one (like C). This means that the source files can be run directly without first creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also interactive.
- **iterable** - A container object capable of returning its members one at a time. Examples of iterables include all sequence types (list, str, tuple, etc.) and some non-sequence types like dict and file and objects of any classes you define with an **iter** or **getitem** method. Iterables can be used in a for loop and in many other places where a sequence is needed (zip(), map(), ...). When an iterable object is passed as an argument to the builtin function iter(), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() or deal with iterator objects yourself - the for statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also iterator, sequence, generator and reiterable.
- **iterator** - An object representing a stream of data. Repeated calls to the iterator's next() method return successive items in the stream. When no more data is available a StopIteration exception is raised instead. At this point the iterator object is exhausted and any further calls to its next() method just raise StopIteration again. Iterators are required to have an **iter**() method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code that attempts multiple iteration passes. A container object (e.g. a list) produces a fresh new iterator each time you pass it to the iter() function or use it in a for loop. Attempting this with an iterator will just return the same exhausted iterator object from the second iteration pass and on, making it appear like an empty container.
- **list comprehension** - A neat syntactical way to process elements in a sequence and return a list with the results. result = ["0x%02x" % x for x in range(256) if x % 2 == 0] generates a list of strings containing hex numbers (0x..) that are even and in the range from 0 to 255. The if part is optional' all elements are processed when it is omitted.
- **list** - A built-in Python datatype, which is a mutable sorted sequence of values. Note that only sequence itself is mutable; it can contain immutable values like strings and numbers. Any Python first-class object can be placed in a tuple as a value.
- **mapping** - A container object (such as dict) that supports arbitrary key lookups using **getitem**.
- **metaclass** - The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.
- **method** - a function defined within a class. Think of this as a "tool" to use against an object (objects are created by classes).
- **mutable** - Mutable objects can change their value but keep their id(). See also immutable.
- **namespace** - The place where a variable is stored in a Python program's memory. Namespaces are implemented as a dictionary. There are the local, global and builtins namespaces and the nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, **builtins**.open() and os.open() are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which modules implement a function. For instance, writing random.seed() and itertools.izip() will make it clear that those functions are implemented by the random and itertools modules respectively.
- **nested scope** - The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.
- **object oriented** - Programming typified by a data-centered (as opposed to a function-centered) approach to program design.
- **object** - Any data with state (attributes or value) and defined behavior (methods).
- **pie syntax** - A syntax using '@' for decorators that was committed to an alpha version of Python 2.4. So called because the '@' vaguely resembles a pie and the committal came on the heels of the Pie-thon at an open source conference in 2004.
- **property** - a built-in data type, used to implement managed (computed) attributes. You assign the property object created by the call property(optional-args) to a class attribute. When the attribute is accessed through an instance of the class, it dispatches functions that implement the managed-attribute operations, such as get-the-value and set-the-value.
- **regular expression** - A formula for matching strings that follow some pattern. Regular expressions are made up of normal characters and metacharacters. In the simplest case, a regular expression looks like a standard search string. For example, the regular expression "testing" contains no metacharacters. It will match "testing" and "123testing" but it will not match "Testing". Metacharacters match some expressions like '.' metacharacter match any single character in a search string.
- **reiterable** - An iterable object which can be iterated over multiple times. Reiterables must not return themselves when used as an argument to iter().
- **sequence** - An iterable that also supports random access using **getitem** and len. Some builtin sequence types are list, str, tuple, and unicode. Note that dict also supports these two operations but is considered a mapping rather than a sequence because the lookups use arbitrary keys rather than consecutive numbers and it should be considered unsorted.
- **static typing** - A style of typing of variables common to many programming languages (such as C) where a variable, having been assigned an object of a given type, cannot be assigned objects of different types subsequently.
- **string** - One of the basic types in Python that store text. In Python 3, stores text as a sequence of Unicode code points.
- **triple-quoted string** - A string that is bounded by three instances of either the double quote mark (") or the single quote mark ('). They are useful for multiple reasons: they allow you to include both single and double quotes within a string quite easily, and they can span multiple lines without the use of line-continuation characters (very useful in docstrings).
- **tuple** - (pronounced TUH-pul or TOO-pul) A built-in Python datatype, which is an immutable ordered sequence of values. Note that only the sequence itself is immutable. If it contains a mutable value such as a dictionary, that value's content may be changed (e.g. adding new key/value pair). Any Python first-class object can be placed in a tuple as a value.
- **type** - A "sort" or "category" of data that can be represented by a programming language. Types differ in their properties (such as mutability and immutability), the methods and functions applicable to them, and in their representations. Python includes, among others, the string, bytes, integer, long, floating point, list, tuple, and dictionary types.
- **whitespace** - The unconventional use of space characters (' ') to control the flow of a program. Instead of a loosely-enforced ideal, this is an integral part of Python syntax. It's a tradeoff between readability and flexibility in favor of the former. first-class object A first class object in a programming language is a

language object that can be created dynamically, stored in a variable, passed as a parameter to a function and returned as a result by a function (from http://www.cs.unm.edu/~crowley/phdExams/1997xfall/pl.html). In Python, practically all objects are first-class, including functions, types, and classes.

57. Designing and Building Our Own API

API Design Best Practice

Most modern APIs are designed to use HTTP as the protocol, so it is worth going over the most common HTTP methods that we can use.

CRUD	Method	Description
Create	POST	Create new resources
Read	GET	Retrieve a representation of a resource
Update	PUT	Update existing resources
Delete	DELETE	Delete existing resources

Developers like to use these HTTP Methods as *verbs* which make things happen on the resource paths. So best practice for designing a URL is to have it include *nouns*, such as `/pets` or `/users`. Then a developer can effectively say `GET /users/<username>` or `DELETE /users/<username>`. This allows the developer to use the HTTP Method as the directive on a specific resource and be very clear in what they are trying to do.

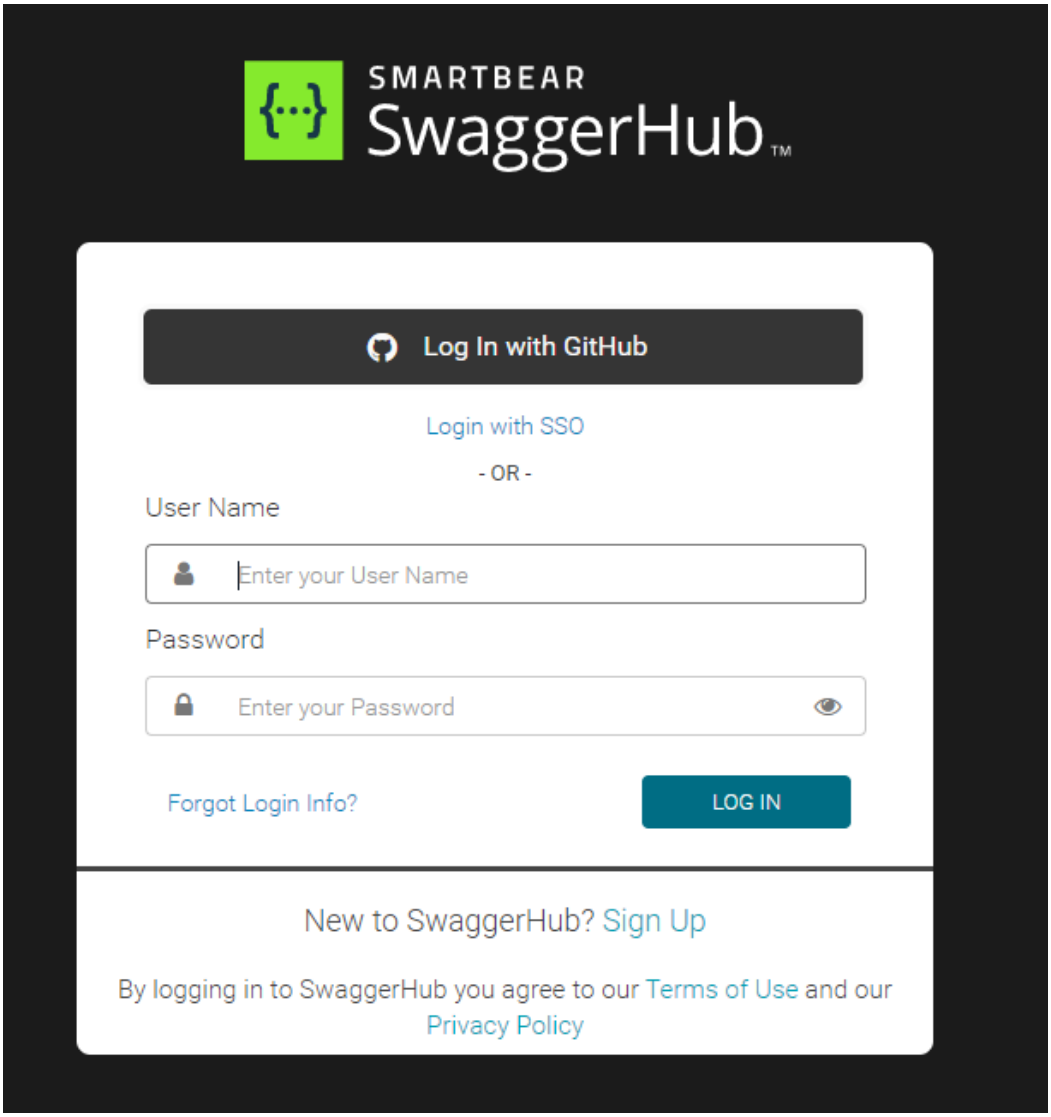
Another important aspect of a well designed API is to always have examples. That way a developer can very quickly understand what an appropriate HTTP request should look like, and also what the expected response would be.

Goals

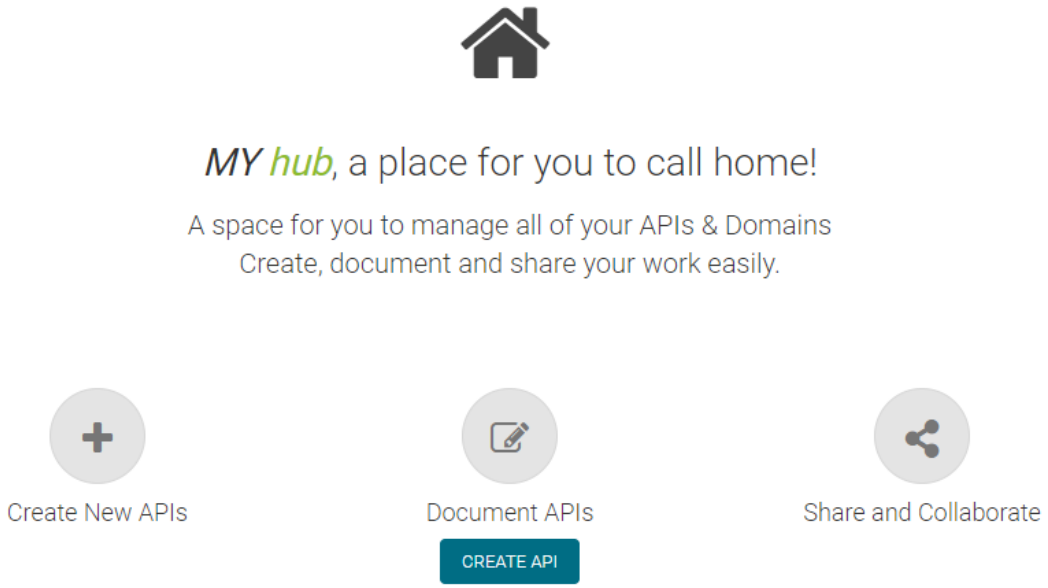
- Use SwaggerHub to **design** our API
- Create an API that allows us to **Create**, **Read**, **Update**, and **Delete** items from a restaurant menu
- Show how to use the aiohttp framework
- Show how to make database calls using aiosqlite

Procedure

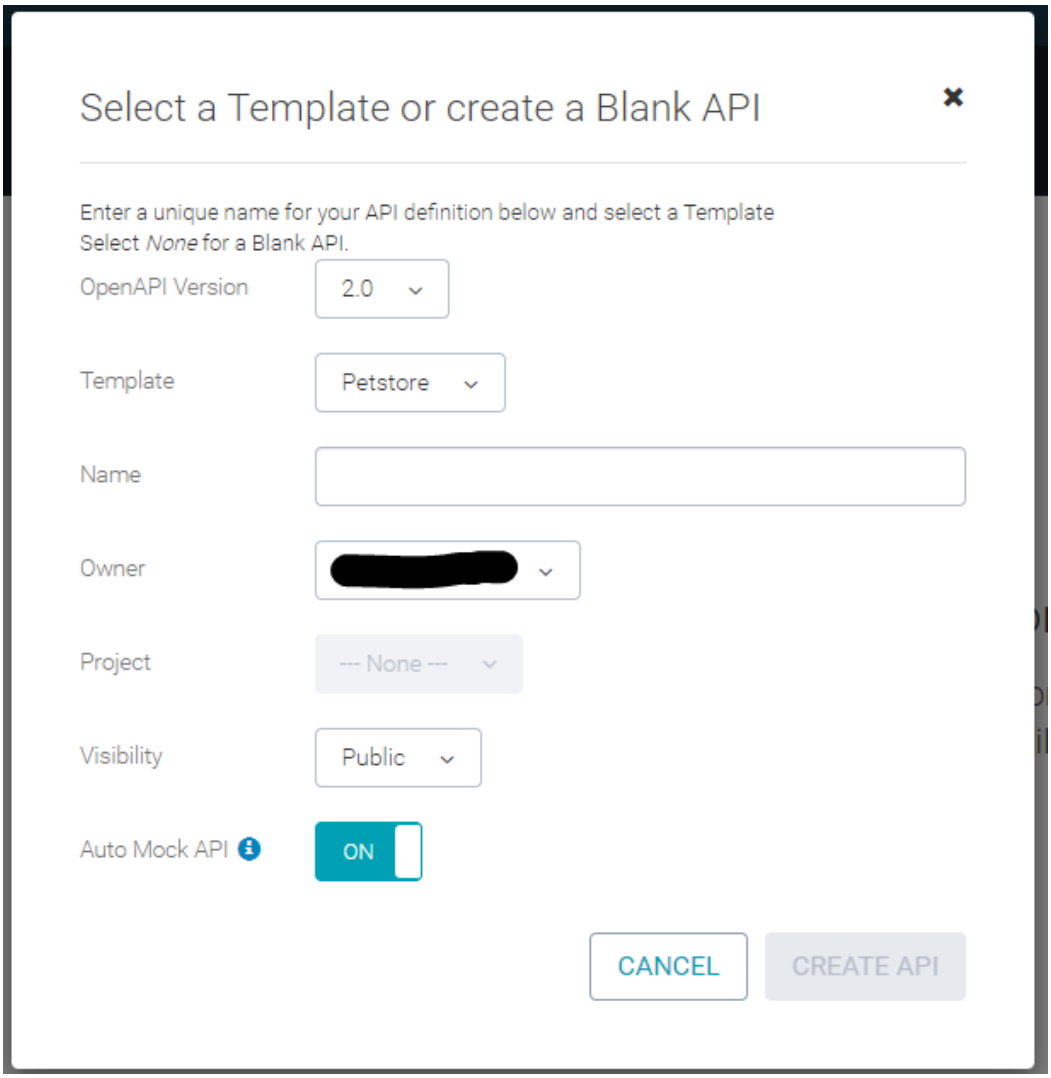
1. In your browser, navigate to https://swagger.io/. Then click on the top right where it says "Log In". This will give you the option to Log in with GitHub, SSO, or SwaggerHub sign-in. We suggest you use your GitHub login.



2. Next you should be greeted by a page like this:



3. Click on the **CREATE API** button. You will then see a screen like this:



Keep all of the settings as seen here, and give it your own name. Then click on the **CREATE API** button.

4. After a few seconds, you should see a new page pop up. The center will have an area to edit the swagger.yaml file which has the design of the API laid out for you. Let's remove all of the YAML text.

DELETE THE YAML CONTENTS

5. Now we will add in our data. First, add in some meta information about this API. **PLEASE MAKE SURE YOU UPDATE YOUR `email` FIELD.**

```
openapi: 3.0.0
servers:
  # Added by API Auto Mocking Plugin
  - description: SwaggerHub API Auto Mocking
    url: https://virtserver.swaggerhub.com/sgriffith3/student-flask-api/1.0.0
info:
  description: This is a menu microservice API
  version: "1.0.0"
  title: Menu Microservice API
  contact:
    # Put your email here!
    email: <me@email.com>
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
tags:
  - name: developers
    description: Operations available to regular developers
```

6. Just after that, let's add a **POST** method to our API. This acts as our **C**reate portion of our CRUD operations.

```
paths:
  # curl -x POST myserver.com/menu ...
  /menu:
    post:
      tags:
        - developers
      summary: adds an inventory item
      operationId: addInventory
      description: Adds an item to the system
      responses:
        '201':
          description: item created
        '400':
          description: 'invalid input, object invalid'
        '404':
          description: page not found
        '409':
          description: an existing item already exists
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/MenuItem'
        description: Inventory item to add
```

7. After this, add in the **GET** method for the **/menu** path. This acts as the **R**ead portion of our CRUD operations.

```
# perform a GET request
get:
  # any special tags we may want to group methods together
  tags:
    - developers
  summary: searches menu
  operationId: get_menu
  description: |
    By passing in the appropriate options, you can search for
    available menu items in the system
  # options that must/may be included in the query string
  parameters:
    - in: query
      name: searchString
      description: pass an optional search string for looking up inventory
      required: false
      schema:
        type: string
  responses:
    # available HTTP responses
    '200':  # 200 == OK
      description: search results matching criteria
      # what the body of the response will look like
      content:
        # specifies the format will be JSON
        application/json:
          schema:
            type: array
            # structure of the data being returned
            items:
              $ref: '#/components/schemas/MenuItem'
    '400':  # 400 == Bad Request
      description: bad input parameter
```

8. Next we will need to allow our users a way to **Update** the existing resources. So let's add in a **PUT** method.

```
put:
  tags:
  - developers
  summary: updates a menu item
  operationId: updateMenu
  responses:
    '200':
      description: item updated
    '400':
      description: invalid object
  requestBody:
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/MenuItem'
    description: Menu item to update
```

9. And finally, we should give our users a way to remove entries from the menu, so we will add in a **DELETE** method.

```
delete:
  tags:
  - developers
  summary: removes a menu item
  operationId: delete_item
  responses:
    '200':
      description: item deleted
    '400':
      description: invalid object
  parameters:
    - in: query
      name: item
      description: pass a string of the item to be deleted
      required: true
      schema:
        type: string
      example: ?item=Kung%20Pao%20Beef
```

10. And finally, we will need to create a definition of our **MenuItem** schema that we have used already in our **put** and **post** methods.

```
components:
  schemas:
    MenuItem:
      type: object
      required:
        - item
        - description
        - price
      properties:
        item:
          type: string
          example: Kung Pao Chicken
        description:
          type: string
          example: Yummy chicken ready to karate kick your tongue
        price:
          type: number
          example: 12.99
```

11. Create the directory called `menu_api`.

```
student@bchd:~$ cd /home/student/mycode && mkdir -p menu_api
```

12. Next, let's create a `requirements.txt` file for our python scripts.

```
student@bchd:~/mycode$ vim menu_api/requirements.txt
```

```
aiohttp
aiosqlite
```

13. Now let's make sure that all of our required third party packages get installed.

```
student@bchd:~/mycode$ python3 -m pip install -r menu_api/requirements.txt
```

14. First, let's create an **admin task as a one-off process**. This script will allow us to create the initial database and table called **menu**. Check out the 12 Factor App to learn about microservice application best practices.

```
student@bchd:~/mycode$ vim menu_api/db_create.py
```

```
import aiosqlite
import asyncio
import os

DB_FILE = os.getenv("DB_FILE", "menu.db")

async def create_menu() -> bool:
    """
    This will create the menu table in the DB_FILE
    """
    async with aiosqlite.connect(DB_FILE) as db:
        sql = "CREATE TABLE IF NOT EXISTS MENU (item CHAR(50), description CHAR(250), price REAL);"
        await db.execute(sql)
    return True


if __name__ == "__main__":
    asyncio.run(create_menu())
```

15. Let's create the database now by running our script.

```
student@bchd:~/mycode$ python3 menu_api/db_create.py
```

16. Now we will need to create our python code to serve actually host our API that will interact with the database.

```
student@bchd:~/mycode$ vim menu_api/menu.py
```

```
import json
import os

from aiohttp import web
import aiosqlite

HOST = os.getenv("MENU_HOST", "0.0.0.0")
PORT = os.getenv("MENU_PORT", 2227)
DB_FILE = os.getenv("DB_FILE", "menu.db")


def routes(app: web.Application) -> None:
    """
    These are the paths that may be appended to the URL.
    Each one has a Request type
    (get == GET, post == POST, put == PUT, delete == DELETE)
    and a different function is called for each type of request
    """

    app.add_routes(
        [
            web.get("/", menu),
            web.get("/menu", menu),
            web.put("/menu", update_menu),
            web.post("/menu", add_item),
            web.delete("/menu", delete_item)
        ]
    )


async def read_menu(search_item=None):
    """Read all of the information from the menu table of the database and return it as an iterable"""
    async with aiosqlite.connect(DB_FILE) as db:
        if search_item:
            sql = f"SELECT * FROM menu where item like '{search_item}'"
        else:
            sql = f"SELECT * FROM menu"
        data = await db.execute(sql)
        return await data.fetchall()


async def menu(request) -> json:
    """
    This will select everything from the menu table in the DB_FILE and return a JSON based web response
    """
    print(request)
    search = request.query.get('item')
    print(search)
    data = await read_menu(search)
    foods = []
    for food in data:
        foods.append({"item": food[0], "description": food[1], "price": food[2]})
    return web.json_response(foods)

async def add_item(request: web.Request) -> web.Response:
    print(request)
    post = await request.json()
    print(post)
    item = post['item']
    desc = post['description']
    price = post['price']
    sql = f"INSERT INTO MENU (item, description, price) VALUES ('{item}', '{desc}', {price});"
    async with aiosqlite.connect(DB_FILE) as db:
        await db.execute(sql)
        await db.commit()
        return web.Response(body="Successfully Updated the Database")


async def update_menu(request: web.Request) -> web.Response:
    print(request)
    put = await request.json()
    print(put)
    item = put['item']
    desc = put['description']
    price = put['price']
    data = await read_menu()
    sql = ""
    for row in data:
        print(row)
        if item == row[0]:
            # If this item already exists, update the description and price
            sql = f"UPDATE MENU SET description = '{desc}', price = {price} where item like '{item}';"
        elif desc == row[1]:
            # If the description matches exactly, update the item name and the price
```

```
                sql = f"UPDATE MENU SET item = '{item}', price = {price} where description like '{desc}';"
            else:
                # If the item and desc do not exist, this is a new item to be added to the database
                sql = f"INSERT INTO MENU (item, description, price) VALUES ('{item}', '{desc}', {price});"
        if sql != "":
            async with aiosqlite.connect(DB_FILE) as db:
                await db.execute(sql)
                await db.commit()
                return web.Response(body="Successfully Updated the Database")

async def delete_item(request: web.Request) -> web.Response:
    """
    This function will read the 'item' query string and attempt to
    DELETE the value from the database
    """
    item = request.query.get('item')
    print(f"Trying to delete {item}")
    sql = f"DELETE FROM MENU WHERE item = '{item}'"
    async with aiosqlite.connect(DB_FILE) as db:
        await db.execute(sql)
        await db.commit()
        return web.Response(body=f"Successfully Deleted {item} from the database")


def main():
    """
    This is the main process for the aiohttp server.

    This works by instantiating the app as a web.Application(),
    then applying the setup function we built in our routes
    function to add routes to our app, then by starting the async
    event loop with web.run_app().
    """

    print("This aiohttp web server is starting up!")
    app = web.Application()
    routes(app)
    web.run_app(app, host=HOST, port=PORT)


if __name__ == "__main__":
    main()
```

17. Now it is time to run the web application.

```
student@bchd:~/mycode$ python3 menu_api/menu.py
```

18. Open (or switch to) another TMUX pane

```
Ctrl b %
```

19. Perform a GET request against our API.

```
student@bchd:~/mycode$ curl localhost:2227/menu
```

> This should currently be a blank list as a response. We have not added anything into our menu yet.

20. Let's add in some data to our menu using curl now. Note that the flag `-X` allows us to select which HTTP Method we wish to use (`POST` in this case), and the flag `-d` allows us to pass data in.

```
student@bchd:~/mycode$ curl -X POST -d '{"item": "Kung Pao Shrimp", "description": "Tasty shrimp ready to kick", "price": 9.88}' localhost:2227/menu

Successfully Updated the Database
```

21. And we should verify that our shrimp was in fact added in.

```
student@bchd:~/mycode$ curl localhost:2227/menu

[{"item": "Kung Pao Shrimp", "description": "Tasty shrimp ready to kick", "price": 9.88}]
```

22. Excellent! Let's add some chicken to our menu too!

```
student@bchd:~/mycode$ curl -X POST -d '{"item": "Teriyaki Chicken", "description": "Best chicken ever!", "price": 11.99}' localhost:2227/menu

Successfully Updated the Database
```

23. And we should also verify that our chicken and shrimp are both in the menu database now.

```
student@bchd:~/mycode$ curl localhost:2227/menu
```

```
[{"item": "Kung Pao Shrimp", "description": "Tasty shrimp ready to kick", "price": 9.88}, {"item": "Teriyaki Chicken", "description": "Best chicken ever!", "price": 11.99}]
```

24. Let's pretend we are now having a sale. The chicken price has been discounted to $10.01 for national palindrome day. We want to update the price without changing anything else about the item, so we can use a `PUT` Request.

```
student@bchd:~/mycode$ curl -X PUT -d '{"item": "Teriyaki Chicken", "description": "Best chicken ever!", "price": 10.01}' localhost:2227/menu

Successfully Updated the Database
```

25. Verify that the chicken price has been updated now.

```
student@bchd:~/mycode$ curl localhost:2227/menu

[{"item": "Kung Pao Shrimp", "description": "Tasty shrimp ready to kick", "price": 9.88}, {"item": "Teriyaki Chicken", "description": "Best chicken ever!", "price": 10.01}]
```

26. Next let's pretend that our restaurant has been badly hurt by a national shrimp shortage, and we can no longer sell our Kung Pao Shrimp. We need to remove it (`DELETE`) from the menu.

```
student@bchd:~/mycode$ curl -X DELETE localhost:2227/menu?item=Kung%20Pao%20Shrimp
```

> Note that any space characters in a URL must be turned into a `%20`

```
Successfully Deleted Kung Pao Shrimp from the databas
```

27. Now let's verify that the shrimp has been removed from the menu.

```
student@bchd:~/mycode$ curl localhost:2227/menu

[{"item": "Teriyaki Chicken", "description": "Best chicken ever!", "price": 10.01}]
```

Awesome work!

You have just designed an API using SwaggerHub (OpenAPI Specifications) and created the code that actually is the API! Way to go!

58. Free Trial Access to Follow-on Courses

You Have Completed:

Python 201: APIs AND API DESIGN WITH PYTHON

Send a FREE 7 day APIs and API Design with Python Course Trial to a colleague:

Free Course Trial!

- **5 Day Course**
- **Lecture and Lab**
- **Every course includes the opportunity to earn an API Design with Python certification from Alta3 Research.**

Course Overview

Application Programming Interfaces (APIs) have become increasingly important as they provide developers with connectivity to everything from rich datasets in an array of formats (such as JSON) to exposing the configurability of software applications and network appliances. Lessons and labs focus on using Python to interact, design, and build APIs for the purposes of scripting automated solutions to complex tasks. Class is a combination of live demonstrations and hands-on labs.

Follow-on Courses:

Python 202: PYTHON FOR NETWORK AUTOMATION

Free Course Trial!

- **5 Day Course**
- **Lecture and Lab**

Course Overview

Managing networks can be repetitive and error-prone, but Python can make incredible changes to how you automate with all major (and most minor) network vendors. This course is driven by lessons and labs that utilize Python libraries designed to interact with and configure your network devices. At the conclusion of this course, you'll be empowered with the tools and skills necessary to take your network to the next level. This class is a combination of live demonstrations and hands-on labs with virtual network devices and endpoints as targets for your configuration.

Ansible 201: NETWORK AUTOMATION WITH PYTHON AND ANSIBLE

Free Course Trial!

- **5 Day Course**
- **Lecture and Hands-On Labs**
- **Certification Project**

Course Overview

Continue your studies of Ansible with a focus on automating common elements within the network. In addition to Ansible, students will study enough Python to understand Ansible's plugin architecture. Lessons and labs focus on using both Python and Ansible to interact with and configure your network devices. At the conclusion of this course, you will return to work empowered with skills necessary to automate network management. This class is a combination of live demonstrations and hands-on labs with virtual network devices and endpoints as targets for your configuration.

Students looking for server applications should see: Ansible 202: Server Automation with Python and Ansible

Moraa Onwonga
moraa.onwonga@accenturefederal.com

59. Lecture - Introduction to threads

Module Objective

The objective of this module is to introduce threads within Python.

Procedures

Python Threading **© Alta3 Research**

Intro to Threading with Python

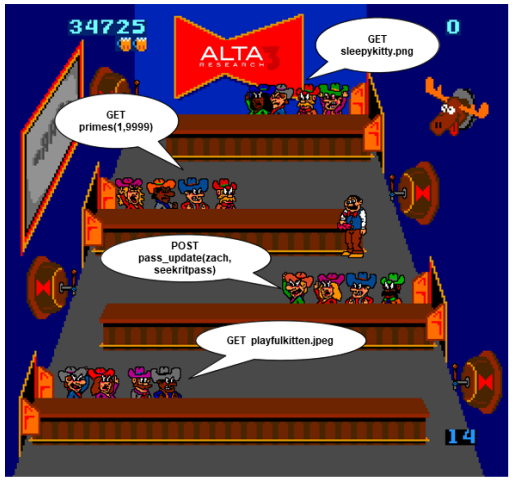
Welcome to the Python Webserver Saloon
"We have ice cold RESTful APIs on tap!"

Threading candidates
- Tasks waiting for external events
- HTTP, Radius, Diameter, and SMTP client / Servers
- ... any server ...

Multiple bars = Multiple threads
- Normally one bar, multiples solves blocking issue
- Technique for managing LOTS of incoming requests
- Does NOT speed up service to an individual request

Waiter = CPU or Core
- Python uses a single core by default

Python Libraries
- `threading`
- `concurrent.futures`

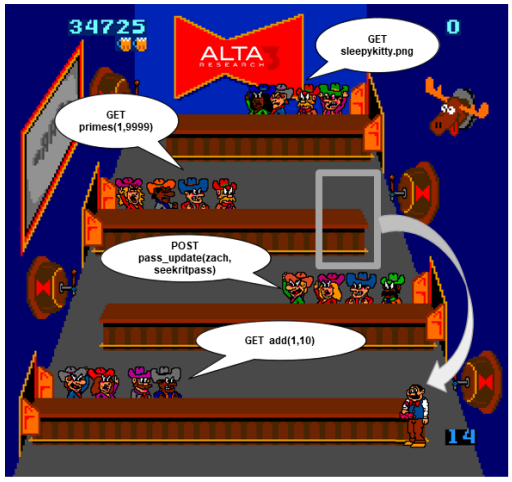


Python Threading **© Alta3 Research**

Context Change

CPU follows to change from one task (or process) to another while ensuring that the tasks do not conflict

Moving between threads
- Think process, request, 'work to do', etc.
- We want to move to happen when *waiting* occurs
- Additionally, Python has a Global Interpreter Lock (GIL) every ~100 bytes the GIL is passed to a new thread
- Store the state of a process or thread so it can be resumed later (*example: waiting for HTTP response)*



Larry Hastings (core Python coder) on Global Interpreter Lock - https://www.youtube.com/watch?v=4zeHStBowEk&t=700s

Moraa Onwonga
moraa.onwonga@accenturefederal.com

Deadlock Errors

One transaction attempts to update a record which has been updated by another transaction which is still active (not committed yet). Normal occurrence in a multi-user programs.

Transaction cause crash or lockup

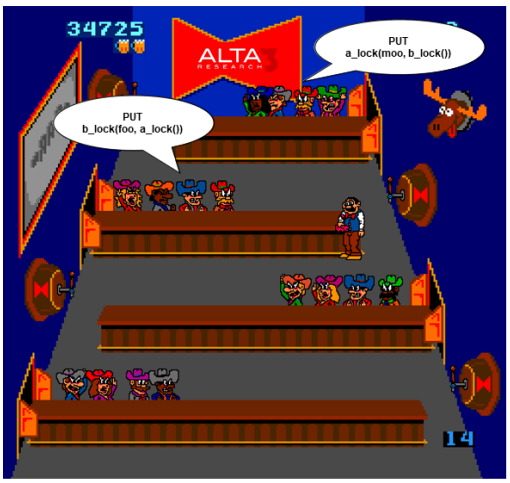
- *Bad – we don't want crashing*
- *Poor design here*

Transaction aborted

- *Quickly deals with issue*
- *Popular design*

Avoided all-together

- *Best*
- *Manual control with threading library*
- *Higher level abstraction with concurrent.futures*



Starvation

A thread is not granted CPU time because other thread(s) grab it all.

- The thread is "starved to death" because other threads are allowed the CPU time instead of it.
- The solution to starvation is called "fairness" - that all threads are fairly granted a chance to execute.

Racing Conditions

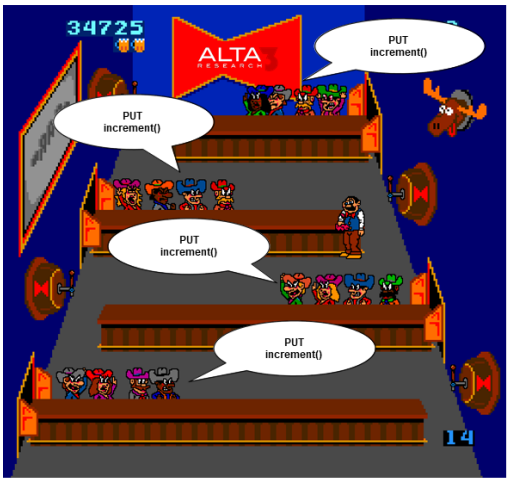
Race conditions are a danger whenever you have more than one process or thread accessing the same data

Example:

- Suppose increment() increases a counter n
- If increment() is called four (4) times concurrently is the result n+1 or n+4

Locks

- Threads that "share state" necessitate locks to prevent this kind of behavior
- Code may be necessary to fully understand...



Moraa Onwonga
moraa.onwonga@accenturefederal.com

Racing – Example (1 of 2)

import db: https://github.com/c-oreills/c-oreills.github.io/tree/master/code/race_conditions

```
1   #!/usr/bin/env python3
2   import db
3
4   def increment():
5       count = db.get_count()
6
7       new_count = count + 1
8       db.set_count(new_count)
9
10      return new_count
11
12
13
14
```

The threading module builds on the low-level features of thread to make working with threads even easier and more pythonic. Using threads allows a program to run multiple operations concurrently in the same process space.

60. Working with Threads

Lab Objective

Threading in Python is used to run multiple threads (tasks, function calls) at the same time. Note that this does not mean that they are executed on different CPUs. Python threads will NOT make your program faster if it already uses 100 % CPU time. In that case, you probably want to look into parallel programming.

Python threads are used in cases where the execution of a task involves some waiting. One example would be interaction with a service hosted on another computer, such as a webserver. Threading allows Python to execute other code while waiting; this is easily simulated with the sleep function.

Procedure

1. Create a new space in which to work.

```
student@bchd:~$ mkdir -p ~/mycode/sewing/
```

2. Move into the directory.

```
student@bchd:~$ cd ~/mycode/sewing/
```

3. Create a new script.

```
student@bchd:~/mycode/sewing$ vim silksuit.py
```

4. Copy and paste the following into your script:

```
#!/usr/bin/python3
"""A basic threading example | rzfeeser@alta3.com"""

# Make a thread that simulates a NASA count down
# waits a 1 seconds at the bottom of each loop

## Python standard library
import threading

## py standard library
import time

def groundcontrol():
    for i in range(10, -1, -1):
        print(i)
        time.sleep(1)

print("Orion you are primed for launch. Count down begins...")

## Create a thread object (target is the function to call)
mythread = threading.Thread(target=groundcontrol)

## begin the thread
mythread.start()
```

5. Save and exit with `:wq`

6. Great! Run your script.

```
student@bchd:~/mycode/sewing$ python3 silksuit.py
```

7. It should work, but it is okay if the purpose of threading is still not clear. Let's write a new script. The purpose is about to become much more clear!

```
student@bchd:~/mycode/sewing$ vim blacktie.py
```

8. At the very bottom of your code, add a few print statements. They can be anything. If you're not feeling creative, copy the example below.

```
#!/usr/bin/python3
"""A basic threading example | rzfeeser@alta3.com"""

# Make a thread that simulates a NASA count down
# Waits 1 second at the bottom of each loop

## Python standard library
import threading

## py standard library
import time

def groundcontrol():
    for i in range(10, -1, -1):
        print(i)
        time.sleep(1)


print("Orion, you are primed for launch. Count down begins...")

## Create a thread object (target is the function to call)
mythread = threading.Thread(target=groundcontrol)

## begin the thread
mythread.start()

## code AFTER we call the thread
print("Uh oh. I forgot my wallet. Can we stop?")
```

9. Save and exit with `:wq`

10. Great! Run your script.

```
student@bchd:~/mycode/sewing$ python3 blacktie.py
```

11. Consider what just happened. Your thread was called **before** your print statement. But your print statement ran **before** your thread finished. How very neat! If you need an analogy, think this way... until now, if you told Python to *"go do my laundry, and then wash the floor,"* Python would start the washing machine, stand there until the washing machine finished, move the clothes to the dryer, stand there until the dryer was finished, and THEN wash the floor. At first we may think, "how very wasteful!" But you might need to tweak that thinking too- instead, think how wonderfully simple that was! It is nice to teach Python to **multitask** but it introduces a new problem that telephony nerds have long understood- that problem is called **racing**. Racing is when a series of steps ends up out of order and it's really bad. For example, consider if we had told Python to multitask between doing your laundry and ironing your laundry. Ironing depends on having clean clothes come out of the dryer first... so best case scenario is Python throws an error, and worst case scenario is Python burns down your house. Let's see if we can mimic this kind of unpredictability.

```
student@bchd:~/mycode/sewing$ vim cleanshirt.py
```

12. Copy and paste the following into your new script. Be sure to read through the comments.

```
#/usr/bin/python3
"""A basic threading example | rzfeeser@alta3.com"""

# Make a thread that simulates a NASA count down
# Waits 1 second at the bottom of each loop

## Python standard library
import threading

## py standard library
import time

def groundcontrol():
    for i in range(10, -1, -1):
        print(i)
        time.sleep(1)

print("Orion, you are primed for launch. Count down begins...")

## Create a thread object (target is the function to call)
mythread = threading.Thread(target=groundcontrol)

## begin the thread
mythread.start()

## Ask the user to press any key to exit.
input("Press Enter to exit.")
exit()
```

13. Save and exit with `:wq`

14. Run your code. Be sure to press `Enter` before you NASA countdown finishes. You'll learn a few things. The first is that your script fails to exit until all threads have finished executing. This is also a basic racing condition as we certainly did not intend to tell the user they can press `Enter` to exit before they can actually exit!

```
student@bchd:~/mycode/sewing$ python3 cleanshirt.py
```

15. We can fix our racing conditions by placing the `.join()` method on our thread. This method tells Python to 'wait' until the thread finishes before moving on. Create a new script.

```
student@bchd:~/mycode/sewing$ vim newshoes.py
```

16. Copy and paste the code below into your new script.

```
#!/usr/bin/python3
"""A basic threading example | rzfeeser@alta3.com"""

# Make a thread that simulates a NASA count down
# Wait 1 second at the bottom of each loop

## Python standard library
import threading

## py standard library
import time

def groundcontrol():
    for i in range(10, -1, -1):
        print(i)
        time.sleep(1)

print("Orion, you are primed for launch. Count down begins...")

## Create a thread object (target is the function to call)
mythread = threading.Thread(target=groundcontrol)

## begin the thread
mythread.start()

print("Oh no, I forgot socks!")

# Wait until the threads finish before moving on.
mythread.join()

## Ask the user to press any key to exit.
input("Press Enter to exit.")
exit()
```

17. Save and exit with `:wq`

18. Run your code. This time, you won't be prompted to press `Enter` until our NASA countdown finishes. That's because the `.join` method prevents our code from moving any further until the script finishes.

```
student@bchd:~/mycode/sewing$ python3 newshoes.py
```

19. We can also have multiple threads at once and let them execute simultaneously. Let's give that a try too. Create another script.

```
student@bchd:~/mycode/sewing$ vim topcoat.py
```

20. To understand the next script, let's deep dive into `import threading`. We use the `thread` library to create an object of Thread class. The Thread class requires the following arguments:

    - `target` - The function to be executed by thread
    - `args` - The arguments to be passed to the target function

21. Copy and paste the following code into your script. Notice that we tweaked the first function to require an argument and added a second function. Now that we have function that requires an argument we'll need to define the `args` parameter.

```
#!/usr/bin/python3
"""A basic threading example | rzfeeser@alta3.com"""

# Make a thread that simulates a NASA count down
# Wait 1 second at the bottom of each loop

## Python standard library
import threading

## py standard library
import time

def groundcontrol(x):
    for i in range(x, -1, -1):
        print(i)
        time.sleep(1)

def orion():
    print("I forgot my socks.")
    time.sleep(1)
    print("Can we stop this ride?")
    time.sleep(2)
    print("No? Alright. Ugh. I forgot to close the garage too.")
    time.sleep(1)
    print("To infinity, and beyond!")

print("Orion, you are primed for launch. Count down begins...")

countdown = 10

## Create a thread object (target is the function to call)
mythread = threading.Thread(target=groundcontrol, args=(countdown, ))

astrothread = threading.Thread(target=orion)

## begin the threads
mythread.start()
astrothread.start()

# Wait until the threads finish before moving on.
mythread.join()
astrothread.join()

## Ask the user to press any key to exit.
input("Press Enter to exit.")
exit()
```

22. Save and exit with `:wq`

23. Run the script. In this simple example we see a kind of 'dueling banjos' back-and-forth between `mythread` and `astrothread`. In a more practical sense, we might use threading to perform multiple database queries before we crunch some numbers.

```
student@bchd:~/mycode/sewing$ python3 topcoat.py
```

24. In another script we can shed a bit more light on how threads work.

```
student@bchd:~/mycode/sewing$ vim tophat.py
```

25. When we create a thread object we can apply a name to that thread object with the `name` parameter. By using that technique, we can print the thread name and corresponding process for each task:

```
#!/usr/bin/python3
""" Naming threads and pids || rzfeeser@alta3.com
Helpful notes:
  - os.getpid() function will return ID of current process
  - threading.main_thread() returns the main thread object.
    Typically the main thread is the thread from which Python was
    started.
  - threading.current_thread() returns the current thread object.
"""


import threading
import os

def task1():
    print("Task 1 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 1: {}".format(os.getpid()))


def task2():
    print("Task 2 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 2: {}".format(os.getpid()))

def main():
    # print ID of current process
    print("ID of process running main program: {}".format(os.getpid()))

    # print name of main thread
    print("Main thread name: {}".format(threading.main_thread().name))

    # creating threads
    t1 = threading.Thread(target=task1, name='t1')
    t2 = threading.Thread(target=task2, name='t2')

    # starting threads
    t1.start()
    t2.start()

    # wait until all threads finish
    t1.join()
    t2.join()

if __name__ == "__main__":
    main()
```

26. Save and exit with `:wq`

27. Run your script.

```
student@bchd:~/mycode/sewing$ python3 tophat.py
```

28. The output should be something like the following:

```
ID of process running main program: 11758
Main thread name: MainThread
Task 1 assigned to thread: t1
ID of process running task 1: 11758
Task 2 assigned to thread: t2
ID of process running task 2: 11758
```

29. The output is proof that a single PID identifies our main process as well as our threads. Interesting!

30. That concludes an overview to multithreading in Python. Even if you don't need to use multithreading, it may help unravel the 'magic' behind an API framework, such as Flask.

31. **CHALLENGE (OPTIONAL)**- Revisit previous code you've written, or perhaps combine previous code you've written. Assign these different functions as threads and observe which complete the fastest. Or, write functions in different ways to see what sort of code runs fastest.

32. If you're tracking your code, run the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "Intro to threading"`
    - `git push origin`

61. Threads and API requests

Lab Objective

The objective of this lab is to learn to apply Python threads to making API requests. Rather than processes these requests sequentially, we want them all to execute seemingly "at the same time".

More truly, it's more like lighting fuses on `n` fireworks.

Traditionally, python wants to light a firework, wait for it's show to complete, before moving onto lighting the next one. This kind of sequential progression will continue until all `n` fireworks have been completed.

However, in some cases, as is with making thousands of API requests as quickly as possible, this kind of *head-of-line-blocking* is not desirable. To invoke our fireworks analogy, threads do not let us create more lighters, but we can change the way Python handles work loads. With threads, we can tell Python to start lighting fuses on fireworks, one after another, and not stop until all `n` fuses have been lit.

Procedure

1. Create a new space in which to work.

```
student@bchd:~$ mkdir -p ~/mycode/apithreads/
```

2. Move into the directory.

```
student@bchd:~$ cd ~/mycode/apithreads/
```

3. Create a new script.

```
student@bchd:~/mycode/apithreads$ vim apiRequestsNoThreads.py
```

4. Copy and paste the following into your script:

```
#!/usr/bin/python3
"""API requests without threads | rzfeeser@alta3.com"""

# standard library
from time import time

# python3 -m pip install requests
import requests

# a list of apis from https://api.le-systeme-solaire.net/rest/bodies/
url_list = [
    "https://api.le-systeme-solaire.net/rest/bodies/lune",
    "https://api.le-systeme-solaire.net/rest/bodies/phobos",
    "https://api.le-systeme-solaire.net/rest/bodies/deimos",
    "https://api.le-systeme-solaire.net/rest/bodies/europe",
    "https://api.le-systeme-solaire.net/rest/bodies/callisto",
    "https://api.le-systeme-solaire.net/rest/bodies/himalia",
    "https://api.le-systeme-solaire.net/rest/bodies/elara",
    "https://api.le-systeme-solaire.net/rest/bodies/sinope",
    "https://api.le-systeme-solaire.net/rest/bodies/leda",
    "https://api.le-systeme-solaire.net/rest/bodies/thebe",
]

def download_file(url):
    html = requests.get(url, stream=True)
    return html.status_code

start = time()

for url in url_list:
    print(download_file(url))

# display the total run time
print(f'Time taken: {time() - start}')
```

5. Save and exit with `:wq`

6. Great! Run your script.

```
student@bchd:~/mycode/apithreads$ python3 apiRequestsNoThreads.py
```

7. The script should complete and show you a run time. An eternity! We can do better...

```
student@bchd:~/mycode/apithreads$ vim apiRequestsThreads.py
```

8. Create the following

```
#!/usr/bin/python3
"""API requests with threads | rzfeeser@alta3.com"""

# standard library
from concurrent.futures import ThreadPoolExecutor, as_completed
from time import time

# python3 -m pip install requests
import requests


url_list = [
    "https://api.le-systeme-solaire.net/rest/bodies/lune",
    "https://api.le-systeme-solaire.net/rest/bodies/phobos",
    "https://api.le-systeme-solaire.net/rest/bodies/deimos",
    "https://api.le-systeme-solaire.net/rest/bodies/europe",
    "https://api.le-systeme-solaire.net/rest/bodies/callisto",
    "https://api.le-systeme-solaire.net/rest/bodies/himalia",
    "https://api.le-systeme-solaire.net/rest/bodies/elara",
    "https://api.le-systeme-solaire.net/rest/bodies/sinope",
    "https://api.le-systeme-solaire.net/rest/bodies/leda",
    "https://api.le-systeme-solaire.net/rest/bodies/thebe",
]

def download_file(url):
    html = requests.get(url, stream=True)
    return html.status_code

start = time()

processes = []

# we want to be careful with the number of workers
# if you are making thousands of requests, does your target have limiting engaged?
# beware you don't overload internal or external services; 5 to 10 is fine for most scripts
with ThreadPoolExecutor(max_workers=5) as executor:
    for url in url_list:
        processes.append(executor.submit(download_file, url))   # add a new task to the threadpool and store in processes list

for task in as_completed(processes):    # yields the items in processes as they complete (it finished or was canceled)
    print(task.result())

# display the total run time
print(f'Time taken: {time() - start}')
```

9. Save and exit with `:wq`

10. Great! Run your script.

```
student@bchd:~/mycode/apithreads$ python3 apiRequestsThreads.py
```

11. The script should execute without error, and show a massive improvement! Answer the following questions:

    - **Q: Why would I read for the concurrent.futures library, as opposed to the threading library?**
        - *A: The threading is a bit more hands on, use it if you need to tightly coordinate state between several threads. If you have no desire to maintain state between threads (such as is with our HTTP requests), concurrent.futures is appropriate.*
    - **Q: If a thread is CPU intensive, will adding threading speed things up?**
        - *A: Generally, no. Python's GIL will allow about 100 bytes of information to process on a thread before the GIL is passed to the next thread (if any). If threads are always waiting for the GIL, adding threads won't speed up your script. However, if your threads do an incredible amount of "waiting" (as is with often with most client and server transactions), threads might be a helpful addition.*
    - **Q: Is concurrent.futures part of the standard library?**
        - *A: Yes. You can find documentation here https://docs.python.org/3/library/concurrent.futures.html*

12. If you're tracking your code, run the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "API requests and threads"`
    - `git push origin`

62. Introduction to Asynchronous Programming with AsyncIO

Lab Objective

The objective of this lab is to learn about AsyncIO library, and its unique way of working.

- **Parallelism** consists of performing multiple operations at the same time. Multiprocessing is a means to effect parallelism, and it entails spreading tasks over a computer's central processing units (CPUs, or cores). Multiprocessing is well-suited for CPU-bound tasks: tightly bound for loops and mathematical computations usually fall into this category. One approach within python is to use the `multiprocessing` package within the standard library.
- **Concurrency** is a slightly broader term than parallelism. It suggests that multiple tasks have the ability to run in an overlapping manner. (There's a saying that concurrency does not imply parallelism.) A way to achieve concurrency is with the `threading`, `concurrent.futures`, and `asyncio` packages.
- **Asynchronous** routines that can be paused while waiting for their ultimate result, to let other routines run. Through this mechanism, it gives the feeling of concurrency. Asynchronous behavior does not require or imply threads or additional processes.
- **Threading** is a concurrent execution model whereby multiple threads take turns executing tasks. One process can contain multiple threads.
- **async IO** is a *single threaded*, *single process* design. It is available within several other languages (Go, Scala, C) however, within Python, is available via the `asyncio` package. The asyncIO model uses *cooperative multitasking* to give a feeling of concurrency despite using a single thread and single process. Coroutines (a central feature of asyncIO) can be scheduled concurrently, but they are not inherently concurrent. It is very much a style of concurrent programming, but it is not parallelism. It's more closely aligned with threading than with multiprocessing but is very much distinct from both of these and is a standalone member in concurrency's bag of tricks.

Resources:

- **async IO** - https://docs.python.org/3/library/asyncio.html
- **AIOHTTP** - https://docs.aiohttp.org/en/latest/

Procedure

1. Answer the following questions:
    - **Q: Is asyncIO part of the Python standard library?**
        - *A: Yes, you can read about its documentation at https://docs.python.org/3/library/asyncio.html*
    - **Q: Will asyncIO make my code multithreaded?**
        - *A: No. Using asyncio in your Python code will not make your code multithreaded. It will not cause multiple Python instructions to be executed at once, and it will not in any way allow you to sidestep the GIL.*
    - **Q: What is a CPU bound process? Is asyncIO used for CPU bound processes?**
        - *A: A process that has a series of instructions needed executed one after another until a result has been achieved. They will make full use of the computer's facilities. AsyncIO is not for CPU bound processes.*
    - **Q: What is a IO-bound process? Will asyncIO help with it?**
        - *A: Yes. In this case, it is fairly common for the CPU to spend a lit of time doing nothing at all because the one thing that's currently being done is waiting for something else.*
    - **Q: What is "best"? Parallelism, concurrency, or threading?**
        - *A: It depends on the application. Generally, parallelism is going to be best for CPU bound tasks, whereas, IO bound tasks are best served with concurrency.*
2. Start in the home directory.

```
student@bchd:~$ cd
```

3. Create a space to work in.

```
student@bchd:~$ mkdir ~/mycode/asyncrequest
```

4. Install `aiohttp`.

```
student@bchd:~$ python3 -m pip install aiohttp
```

5. Create a script that uses `aiohttp` and `asyncio`.

```
student@bchd:~$ vim ~/mycode/asyncrequest/async01.py
```

6. Create the following:

```
#!/usr/bin/env python3
"""RZFeeser | rzfeeser@alta3.com"""

# standard library
import aiohttp
import asyncio

# create a coroutine called 'main'
async def main():           # the async keyword creates a coroutine to be run asynchronously

    async with aiohttp.ClientSession() as session:
        # request the pokemon 'Mew' (pokemon number 151)
        pokemon_url = 'https://pokeapi.co/api/v2/pokemon/151'
        async with session.get(pokemon_url) as resp:
            pokemon = await resp.json()     # passes control back to the event loop suspending execution of coroutine until
                                            # the awaited result is returned
            print(pokemon['name'])

asyncio.run(main())
```

7. Save and exit your script with `:wq`

8. Try running your script. (NOTE: any version of Python before 3.7 will not have `asyncio.run()`)

```
student@bchd:~$ python3 ~/mycode/asyncrequest/async01.py
```

9. Great! So what if we wanted 150 API requests performed in a very rapid maner, with asyncIO. Let's try creating a loop, and running the API look-ups asynchronously.

```
student@bchd:~$ vim ~/mycode/asyncrequest/async02.py
```

10. Create the following:

```
#!/usr/bin/env python3
"""Alta3 Research | RZFeeser
   Demonstrating how to use the asyncio library by utilizing the pokeapi.co
   to perform 150 HTTP GET lookups"""

# standard library
import asyncio
import time

# python3 -m pip install aiohttp
import aiohttp

# start a timer to determine how quickly these lookups are performed
start_time = time.time()

async def main():

    async with aiohttp.ClientSession() as session:
        # loop from 1 to 150 (non inclusive of 151)
        for number in range(1, 151):
            pokemon_url = f'https://pokeapi.co/api/v2/pokemon/{number}'    # number is defined by the range for-loop
            async with session.get(pokemon_url) as resp:     # the coroutine we are defining should be run async with an event loop
                pokemon = await resp.json()         # pass control back to the event loop (do other things until this happens)
                print(pokemon['name'])

asyncio.run(main())
print("--- %s seconds ---" % (time.time() - start_time))
```

11. Save and exit with `:wq`

12. Try running your script.

```
student@bchd:~$ python3 ~/mycode/asyncrequest/async02.py
```

13. Let's compare the result to a synchronous series of requests. This can be achieved with a simple for loop, and the requests library.

```
student@bchd:~$ vim ~/mycode/asyncrequest/async03_no_async.py
```

14. Create the following script:

```
#!/usr/bin/python3
"""Alta3 Research | RZFeeser@alta3.com
   Synchronous requests, this should be much slower than the asynchronous method"""

# standard library
import time

# python3 -m pip install requests
import requests

def main():
    # start a timer
    start_time = time.time()

    # typical loop
    for number in range(1, 151):
        url = f'https://pokeapi.co/api/v2/pokemon/{number}'
        resp = requests.get(url)
        pokemon = resp.json()
        print(pokemon['name'])

    print("--- %s seconds ---" % (time.time() - start_time))

# call our script if it was not imported
if __name__ == "__main__":
    main()
```

15. Save and exit with `:wq`

16. Run the script. Our theory is that this one will take quite a bit *more* time.

```
student@bchd:~$ python3 ~/mycode/asyncrequest/async03_no_async.py
```

17. The script should execute correctly, but notice how much slower this method is. Answer the following questions:

    - **Q: Why was aiohttp required? Why didn't we just use requests?**
        - *A: Within a single thread, requests is a blocking synchronous library. When it is used in conjunction with threads, this does not present a problem, however, if we are working with an asynchronous library, like asyncio, we need a client that behaves aynchronously.*
    - **Q: Where can I read more about aiohttp and the design decisions?**
        - *A: Read more about the aiohttp client and how it differs from requests library here, https://docs.aiohttp.org/en/latest/http_request_lifecycle.html#why-is-aiohttp-client-api-that-way*

18. If you're tracking your code, run the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "making requests with asyncIO"`
    - `git push origin`

63. pandas dataframes with Excel, csv, json, HTML and beyond

Lab Objective

The objective of this lab is to learn how to work with datasets commonly encountered by network engineers. Vendors often will supply data in formats such as Excel (xls and xlsx), csv, json, and plain-text. The Python library `pandas` makes it easy to transform data into a common format (dataframes), which can than be exported to whatever format is required. Other reasons for using dataframes include data analysis, plotting and graphing, moving data to machine learning tools (scikit-learn), or possibly moving data across RESTful API interfaces.

The pandas library tools for reading and writing data between in-memory data structures and different formats include Excel, as well as CSV and text files, SQL databases, and the fast HDF5 format.

Read about the Pandas project here:

https://pandas.pydata.org/

The data types pandas can read and write to can be found here:

https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html

Procedure

1. Open a new terminal
2. Create a new directory to work in.

```
student@bchd:~$ mkdir -p ~/mycode/pandas/
```

3. Move into that directory.

```
student@bchd:~$ cd ~/mycode/pandas/
```

4. Ensure pandas is installed along with `xlrd`, and `openpyxl`, which are optional dependencies for working with MS Excel.

```
student@bchd:~/mycode/pandas$ python3 -m pip install pandas xlrd openpyxl
```

5. Download the Excel data set we'll be using for this lab. The set itself spans 3 sheets, and contains movie data. You may want to download this dataset on your local laptop, and check it out in Excel.

```
student@bchd:~/mycode/pandas$ wget https://static.alta3.com/files/movies.xls
```

6. Create a new script, `pandabear01.py`

```
student@bchd:~/mycode/pandas$ vim pandabear01.py
```

7. Create the following solution:

```
#!/usr/bin/python3

import pandas as pd

def main():
    # define the name of our xls file
    excel_file = 'movies.xls'

    # create a DataFrame (DF) object. EASY!
    # because we did not specify a sheet
    # only the first sheet was read into the DF
    movies = pd.read_excel(excel_file)

    # show the first five rows of our DF
    # DF has 5 rows and 25 columns (indexed by integer)
    print(movies.head())

    # Choose the first column "Title" as
    # index (index=0)
    movies_sheet1 = pd.read_excel(excel_file, sheet_name=0, index_col=0)
    # DF has 5 rows and 24 columns (indexed by title)
    print(movies_sheet1.head())

    # grab the next 2 sheets as well
    movies_sheet2 = pd.read_excel(excel_file, sheet_name=1, index_col=0)
    # DF has 5 rows and 24 columns (indexed by title)
    print(movies_sheet2.head())

    movies_sheet3 = pd.read_excel(excel_file, sheet_name=2, index_col=0)
    # DF has 5 rows and 24 columns (indexed by title)
    print(movies_sheet3.head())

    # combine all DFs into a single DF called movies
    movies = pd.concat([movies_sheet1, movies_sheet2, movies_sheet3])

    # number of rows and columns (5042, 24)
    print(movies.shape)

    # sort DataFrame based on Gross Earnings
    sorted_by_gross = movies.sort_values(["Gross Earnings"], ascending=False)

    # Data is sorted by values in a column
    # display the top 10 movies by Gross Earnings.
    # passing the 10 values to head returns the top 10 not the default 5
    print(sorted_by_gross.head(10))

if __name__ == "__main__":
    main()
```

8. Save and exit.

9. Run your script.

```
student@bchd:~/mycode/pandas$ python3 pandabear01.py
```

10. Let's try combining multiple data sources, CSV and JSON, into a single data frame, then exporting to a few formats, including Excel. Start by downloading a CSV dataset.

```
student@bchd:~/mycode/pandas$ wget https://static.alta3.com/files/ciscodata.csv
```

11. Now, download a JSON dataset.

```
student@bchd:~/mycode/pandas$ wget https://static.alta3.com/files/ciscodata2.json
```

12. Display both datasets. Both have valuable data, but they're in mixed formats. First, display the CSV data.

```
student@bchd:~/mycode/pandas$ cat ciscodata.csv
```

13. Display the json.

```
student@bchd:~/mycode/pandas$ cat ciscodata2.json
```

14. Great. Now write a script that will take those data sets, and combine them into a single dataframe.

15. Create a new script, `pandabear02.py`

```
student@bchd:~/mycode/pandas$ vim pandabear02.py
```

16. Create the following script:

```
#!/usr/bin/python3

import pandas as pd

def main():
    ciscocsv = pd.read_csv("ciscodata.csv")
    ciscojson = pd.read_json("ciscodata2.json")

    # display first 5 entries of the ciscocsv dataframe
    print(ciscocsv.head())

    # display first 5 entries of the ciscojson dataframe
    print(ciscojson.head())

    ciscodf = pd.concat([ciscocsv, ciscojson])
    # uncomment the line below to "fix" the index issue
    # ciscodf = pd.concat([ciscocsv, ciscojson], ignore_index=True, sort=False)

    print(ciscodf)

if __name__ == "__main__":
    main()
```

17. Save and exit.

18. Execute your code.

```
student@bchd:~/mycode/pandas$ python3 pandabear02.py
```

19. Before you go any further, study the results. Notice how the indexes are repeated? There is a fix for this, which is described here: https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html under, "Ignoring indexes on the concatenation axis".

20. Let's rewrite that script. This time, we'll fix the indexing issue, and export our data a few different ways. To export to Excel, you'll need to install `xlwt` with `pip`. Failure to do so will remind you that you need to install `xlwt`.

```
student@bchd:~/mycode/pandas$ python3 -m pip install xlwt
```

21. Create a new script, `pandabear03.py`

```
student@bchd:~/mycode/pandas$ vim pandabear03.py
```

22. Write the following script:

```
#!/usr/bin/python3

import pandas as pd

def main():
    # create a dataframe ciscocsv
    ciscocsv = pd.read_csv("ciscodata.csv")
    # create a dataframe ciscojson
    ciscojson = pd.read_json("ciscodata2.json")

    # The line below concats and reapplies the index value
    ciscodf = pd.concat([ciscocsv, ciscojson], ignore_index=True, sort=False)

    ## print to the screen the re-indexed dataframe
    print(ciscodf)

    ## print a blankline
    print()

    ## export to json
    ciscodf.to_json("combined_ciscodata.json")

    ## export to csv
    ciscodf.to_csv("combined_ciscodata.csv")

    ## export to Excel
    ciscodf.to_excel("combined_ciscodata.xls")

    ## create a python dictionary
    x = ciscodf.to_dict()
    print(x)

if __name__ == "__main__":
    main()
```

23. Save and exit.

24. Execute your code.

```
student@bchd:~/mycode/pandas$ python3 pandabear03.py
```

25. Study the output. First the `combined_ciscodata.json` file.

```
student@bchd:~/mycode/pandas$ cat combined_ciscodata.json
```

26. Now `combined_ciscodata.csv`

```
student@bchd:~/mycode/pandas$ cat combined_ciscodata.csv
```

27. Let the XLS file sit for now.

28. There's a few problems. The JSON data that was produced now has the index numbers included in it, as does the CSV data. Let's see if we can write an improved script that fixes this.

29. Create a new script, `pandabear04.py`

```
student@bchd:~/mycode/pandas$ vim pandabear04.py
```

30. Write the following script:

```
#!/usr/bin/python3

import pandas as pd

def main():
    # create a dataframe ciscocsv
    ciscocsv = pd.read_csv("ciscodata.csv")
    # create a dataframe ciscojson
    ciscojson = pd.read_json("ciscodata2.json")

    # The line below concats and reapplies the index value
    ciscodf = pd.concat([ciscocsv, ciscojson], ignore_index=True, sort=False)

    ## export to json
    ## do not include index number
    ciscodf.to_json("combined_ciscodata.json", orient="records")

    ## export to csv
    ## do not include index number
    ciscodf.to_csv("combined_ciscodata.csv", index=False)

    ## export to Excel
    ## do not include index number to xls
    ciscodf.to_excel("combined_ciscodata.xls", index=False)
    ## do not include index number to xlsx
    ciscodf.to_excel("combined_ciscodata.xlsx", index=False)

    ## create a python dictionary
    ## do not include index number
    x = ciscodf.to_dict(orient='records')
    print(x)

if __name__ == "__main__":
    main()
```

31. Save and exit.

32. Execute your code.

```
student@bchd:~/mycode/pandas$ python3 pandabear04.py
```

33. Study the output which no longer includes the index values. First the `combined_ciscodata.json` file.

```
student@bchd:~/mycode/pandas$ cat combined_ciscodata.json
```

34. Now `combined_ciscodata.csv` Notice, no more index values.

```
student@bchd:~/mycode/pandas$ cat combined_ciscodata.csv
```

35. Download the new dataset, `corporateNetworkLog.csv`. This dataset is a reflection of data usage across a corporate network from 2006 through 2017. The data is arranged, `Date,Consumption,YouTube,Netflix,YouTube+Netflix`. `Date` is given via YYYY-MM-DD. All other values given are in terabytes. The value `Consumption`, is the sum of total network traffic for a small corporate center. The values `YouTube` and `Netflix` are the portion of `Consumption` consumed by either service. The value `YouTube+Netflix` is simply the sum of `YouTube` and `Netflix` for that date.

```
student@bchd:~/mycode/pandas$ wget https://static.alta3.com/courses/pyna/netTraffic.csv
```

36. Originally developed for financial time series such as daily stock market prices, the robust and flexible data structures in pandas can be applied to time series data in any domain, including business, science, engineering, public health, and many others. With these tools you can easily organize, transform, analyze, and visualize your data at any level of granularity — examining details during specific time periods of interest, and zooming out to explore variations on different time scales, such as monthly or annual aggregations, recurring patterns, and long-term trends... such as how much time employees are spending streaming movies.

A time series really could be any data set where the values are measured at different points in time. Many time series are uniformly spaced at a specific frequency, for example, hourly weather measurements, daily counts of web site visits, or cell sites currently up or down. Time series can also be irregularly spaced and sporadic, for example, timestamped data in a computer system's event log or a history of 911 emergency calls. Pandas time series tools apply equally well to either type of time series.

37. With pandas, we can answer the following questions:

    - **When is network consumption typically highest and lowest?**
    - **How do streaming services vary with seasons of the year?**
    - **What are the long-term trends in network usage, YouTube, and Netflix?**
    - **How do YouTube and Netflix compare with network consumption, and how has this ratio changed over time?**

38. To begin, create a new script:

```
student@bchd:~/mycode/pandas$ vim pandabear05.py
```

39. Create the following script:

```
#!/usr/bin/python3
"""Russell Zachary Feeser || Alta3 Research
In pandas, a single point in time is represented as a Timestamp. We can use the to_datetime() function to create Timestamps from strings in a wide variety
of date/time formats. Let's import pandas and convert a few dates and times to Timestamps.
"""

import pandas as pd

def main():
    """run-time code"""
    # to_datetime() automatically infers a date/time format based on the input
    # the ambiguous date '7/8/1952' is assumed to be month/day/year and is interpreted as July 8, 1952
    print(pd.to_datetime('2018-01-15 3:45pm'))
    # 2018-01-15 15:45:00

    # Alternatively, we can use the dayfirst parameter to tell pandas to interpret the date as August 7, 1952.
    print(pd.to_datetime('7/8/1952'))
    # 1952-07-08 00:00:00

    print(pd.to_datetime('7/8/1952', dayfirst=True))
    # 1952-08-07 00:00:00

    # Supply a list or array of strings as input to to_datetime() and it
    # returns a sequence of date/time values in a DatetimeIndex object, which is the core data structure that powers much of pandas time series
functionality
    print(pd.to_datetime(['2018-01-05', '7/8/1952', 'Oct 10, 1995']))
    # DatetimeIndex(['2018-01-05', '1952-07-08', '1995-10-10'], dtype='datetime64[ns]', freq=None)
    # In the DatetimeIndex above, the data type datetime64[ns] indicates that the underlying data is stored as 64-bit integers, in units of nanoseconds
(ns)
    # This data structure allows pandas to compactly store large sequences of date/time values and efficiently perform vectorized operations using NumPy
datetime64 arrays.

    # Dealing with a sequence of strings all in the same date/time format, we can explicitly specify it with the format parameter
    # For very large data sets, this can greatly speed up the performance of to_datetime() compared to the default behavior
    # Any of the format codes from the strftime() and strptime() functions in Python's built-in datetime module can be used.
    print(pd.to_datetime(['2/25/10', '8/6/17', '12/15/12'], format='%m/%d/%y'))
    # DatetimeIndex(['2010-02-25', '2017-08-06', '2012-12-15'], dtype='datetime64[ns]', freq=None)


if __name__ == "__main__":
    main()
```

40. Save and exit with `:wq`

41. Try running your script.

```
student@bchd:~/mycode/pandas$ python3 pandabear05.py
```

42. Now that we know a bit more about pandas and dates, lets creating a time series DataFrame. To do this, we use the read_csv() function to read the data into a DataFrame.

```
student@bchd:~/mycode/pandas$ vim pandabear06.py
```

43. Create the following script that will work with our new dataset.

```
#!/usr/bin/python3
"""Russell Zachary Feeser | Alta3 Research
Creating a DataFrame from a timeseries dataset and working with methods to display various network usage.
"""

import pandas as pd

def main():
    opsd_daily = pd.read_csv('netTraffic.csv') # this opens the csv dataset

    print(opsd_daily.shape)
    # (4383, 5)  # our data has 4383 rows covering January 1, 2006 through December 31, 2017

    print("\nLook at the first three rows")
    print(opsd_daily.head(3))

    print("\nLook at the last three rows")
    print(opsd_daily.tail(3))

    # check out the data types of each column
    print(opsd_daily.dtypes)

    # set the date as the DataFrame's index
    opsd_daily = opsd_daily.set_index('Date')

    print("\nLook at the first three rows after date has been set as the primary index")
    print(opsd_daily.head(3))

    print("\nLook at the last three rows after date has been set as the primary index")
    print(opsd_daily.tail(3))

    # display all of the index values (this is a lot of data)
    input("\nPress ENTER to look at all of the index values associated with the dataset (dates)")
    print(opsd_daily.index)

    # consolidate the above steps into a single line using the index_col and parse_dates parameters of the read_csv() function
    opsd_daily = pd.read_csv('netTraffic.csv', index_col=0, parse_dates=True)

    # add some additional columns to our data
    # Add columns with year, month, and weekday name
    opsd_daily['Year'] = opsd_daily.index.year
    opsd_daily['Month'] = opsd_daily.index.month
    # required to 'pull' the day name (ex. Monday, Tuesday, happy days...)
    opsd_daily['Weekday Name'] = opsd_daily.index.day_name()

    # display a random sampling of 5 rows
    input("\nPress ENTER to look at a random sampling from 5 rows after adding the Year, Month and Weekday Name columns")
    print(opsd_daily.sample(5, random_state=0))


if __name__ == "__main__":
    main()
```

44. Save and exit.

45. Try running your script.

```
student@bchd:~/mycode/pandas$ python3 pandabear06.py
```

46. Now that we can easily create a pandas DataFrame with an index using DatetimeIndex, we can use all of pandas' powerful time-based indexing to wrangle and analyze our data. One of the most powerful and convenient features of pandas time series is time-based indexing — using dates and times to intuitively organize and access our data. Create a new script.

```
student@bchd:~/mycode/pandas$ vim pandabear07.py
```

47. Create the following script:

```
#!/usr/bin/python3
"""Russell Zachary Feeser | Alta3 Research
Learning to work with Time-based indexing in pandas
"""

import pandas as pd

def main():
    """run-time code"""
    # consolidate the above steps into a single line using the index_col and parse_dates parameters of the read_csv() function
    opsd_daily = pd.read_csv('netTraffic.csv', index_col=0, parse_dates=True)

    # add some additional columns to our data
    # Add columns with year, month, and weekday name
    opsd_daily['Year'] = opsd_daily.index.year
    opsd_daily['Month'] = opsd_daily.index.month
    # required to 'pull' the day name (ex. Monday, Tuesday, happy days...)
    opsd_daily['Weekday Name'] = opsd_daily.index.day_name()

    # select data for a single day using a string such as '2017-08-10'
    input("\nPress ENTER to see the data for 2017-08-10")
    print(opsd_daily.loc['2017-08-10'])

    # select a slice of days, '2014-01-20':'2014-01-22'
    # Note that the slice is inclusive of both endpoints
    input("\nPress ENTER to see the data slice from 2014-01-20 to 2014-01-22")
    print(opsd_daily.loc['2014-01-20':'2014-01-22'])

    # partial-string indexing select all date/times which partially match a given string
    # select the entire year 2006 with opsd_daily.loc['2006']
    # select the entire month of February 2012 with opsd_daily.loc['2012-02']
    input("\nPress ENTER to see the data slice for 2012-02")
    print(opsd_daily.loc['2012-02'])

if __name__ == "__main__":
    main()
```

48. Save and exit

49. Run the new script.

```
student@bchd:~/mycode/pandas$ python3 pandabear07.py
```

50. With pandas, and a second library, matplotlib, we can easily visualize our time series data. First, install matplotlib.

```
student@bchd:~/mycode/pandas$ python3 -m pip install matplotlib
```

51. Now install seaborn. Seaborn is a library for making statistical graphics in Python. It is built on top of matplotlib and closely integrated with pandas data structures. Read about the project here: https://pypi.org/project/seaborn/

```
student@bchd:~/mycode/pandas$ python3 -m pip install seaborn
```

52. Create a new script.

```
student@bchd:~/mycode/pandas$ vim pandabear08.py
```

53. Create the following script:

```
#!/usr/bin/python3
"""Russell Zachary Feeser | Alta3 Research
Learning to work with Time-based indexing in pandas
"""

import pandas as pd

import matplotlib
matplotlib.use('Agg') # required to generate images without a window appearing

# does not appear to be necessary
#import matplotlib.pyplot as plt

import seaborn as sns

def main():
    """run-time code"""
    # consolidate the above steps into a single line using the index_col and parse_dates parameters of the read_csv() function
    opsd_daily = pd.read_csv('netTraffic.csv', index_col=0, parse_dates=True)

    # add some additional columns to our data
    # Add columns with year, month, and weekday name
    opsd_daily['Year'] = opsd_daily.index.year
    opsd_daily['Month'] = opsd_daily.index.month
    # required to 'pull' the day name (ex. Monday, Tuesday, happy days...)
    opsd_daily['Weekday Name'] = opsd_daily.index.day_name()

    # Use seaborn style defaults and set the default figure size
    sns.set(rc={'figure.figsize':(11, 4)})


    ### LINE PLOT - create a line plot of the full time series of daily network consumption, using the DataFrame's plot() method.
    netlineplot = opsd_daily['Consumption'].plot(linewidth=0.5)

    # save out this figure
    fig = netlineplot.get_figure()
    fig.savefig("/home/student/static/linePlot.png")



    ### DOT PLOT - plot the data as dots instead, and also look at the YouTube and Netflix time series
    cols_plot = ['Consumption', 'YouTube', 'Netflix']
    axes = opsd_daily[cols_plot].plot(marker='.', alpha=0.5, linestyle='None', figsize=(11, 9), subplots=True)

    for ax in axes:
        ax.set_ylabel('Daily Totals (TBs)')

        # save out this figure
        fig = ax.get_figure()
        fig.savefig(f"/home/student/static/dotPlot.png")

if __name__ == "__main__":
    main()
```

54. Save and exit.

55. Try running your code:

```
student@bchd:~/mycode/pandas$ python3 pandabear08.py
```

56. The graphs were saved out to the `~/static/` folder. Files in this folder can be viewed by clicking on the icon that is 3-sheets of paper in the upper right hand corner, and then selecting your instance of `files`. By using this method, you should see two files, `linePlot.png` and `dotPlot.png`. Try interacting (clicking) on both.

57. Looks like some interesting patterns are emerging! The data suggests that there may be some seasonality, and weekly changes in network consumption corresponding with weekdays and weekends.

1. If you're tracking your code in an SCM platform, issue the following commands:
    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "working with pandas"`
    - `git push origin HEAD`

64. Paramiko - SFTP with UN and PW

Lab Objective

The objective of this lab is to learn to push files across an SFTP connection using Paramiko. This lab uses variations of a password/username combination. However, Paramiko can also work with a RSA keypair.

Procedure

1. We'll be using the **planetexpress** team for this lab! Run the command below to prepare your environment.

```
student@bchd:~$ bash ~/px/scripts/full-setup.sh
```

2. Paramiko works across SSH tunnels, so we'll want to make sure that is installed. This is supported by Python 2.7+ and 3.x releases, so everyone should feel at home using it. Read the documentation page here: http://www.paramiko.org/installing.html

```
student@bchd:~$ python3 -m pip install paramiko
```

3. Make the directories we will need:

```
student@bchd:~$ mkdir -p ~/filestocopy/ ~/mycode/paramikosftp
```

4. Make a empty file for us to copy:

```
student@bchd:~$ touch /home/student/filestocopy/myfile.txt
```

5. Make another empty file for us to copy:

```
student@bchd:~$ touch /home/student/filestocopy/myfile02.txt
```

6. Make a third empty file for us to copy:

```
student@bchd:~$ touch /home/student/filestocopy/myfile03.txt
```

7. Move into the directory where we will write our code.

```
student@bchd:~$ cd ~/mycode/paramikosftp/
```

8. Create a new script.

```
student@bchd:~/mycode/paramikosftp$ vim ~/mycode/paramikosftp/sftpmover.py
```

9. Add the following into your script:

```
#!/usr/bin/env python3
"""Alta3 Research | RZFeeser@alta3.com
   Moving files with SFTP"""

## import paramiko so we can talk SSH
import paramiko
import os

## where to connect to
t = paramiko.Transport("10.10.2.3", 22) ## IP and port

## how to connect (see other labs on using id_rsa private/public keypairs)
t.connect(username="bender", password="alta3")

## Make an sftp connection object
sftp = paramiko.SFTPClient.from_transport(t)

## iterate across the files within directory
for x in os.listdir("/home/student/filestocopy/"): # iterate on directory contents
  if not os.path.isdir("/home/student/filestocopy/"+x): # filter everything that is NOT a directory
    sftp.put("/home/student/filestocopy/"+x, "/tmp/"+x) # move file to target location

## close the connections
sftp.close() # close the connection
t.close()
```

10. Save and close, then run the script.

```
student@bchd:~/mycode/paramikosftp$ python3 ~/mycode/paramikosftp/sftpmover.py
```

11. SSH to the bender container.

```
student@bchd:~/mycode/paramikosftp$ ssh bender@10.10.2.3
```

12. List the contents of `/tmp/`

```
bender@bender:~$ ls /tmp/
```

13. Exit the bender machine.

```
bender@bender:~$ exit
```

14. Create a new script.

```
student@bchd:~/mycode/paramikosftp$ vim learning_getpass.py
```

15. Copy and paste the following into your script:

```
#!/usr/bin/python3
# A simple Python program to demonstrate  getpass.getpass() to read password
import getpass

def main():
    p = getpass.getpass()
    print("Password entered:", p)

if __name__ == "__main__":
    main()
```

16. Run this very basic script, and make sure that it works. Try entering **alta3** as your password.

```
student@bchd:~/mycode/paramikosftp$ python3 learning_getpass.py
```

17. Create a new script.

```
student@bchd:~/mycode/paramikosftp$ vim sftpmover_getpass.py
```

18. Copy and paste the following into your script:

```
#!/usr/bin/python3
## Try a real world test with getpass

## import Paramiko so we can talk SSH
import paramiko # allows Python to ssh
import os # low level operating system commands
import getpass # we need this to accept passwords


def main():
    ## where to connect to
    t = paramiko.Transport("10.10.2.3", 22) ## IP and port of bender

    ## how to connect (see other labs on using id_rsa private / public keypairs)
    t.connect(username="bender", password=getpass.getpass()) # notice the password references getpass

    ## Make an SFTP connection object
    sftp = paramiko.SFTPClient.from_transport(t)

    ## copy our firstpasswd.py script to bender
    sftp.put("file_to_move.txt", "file_to_move.txt") # move file to target location home directory

    ## close the connection
    sftp.close() # close the connection
if __name__ == "__main__":
    main()
```

19. Save and exit with `:wq`

20. Create a file we can transfer.

```
student@bchd:~/mycode/paramikosftp$ touch file_to_move.txt
```

21. Save and close, then run the script (see the next step, you'll need to enter the password *alta3*).

```
student@bchd:~/mycode/paramikosftp$ python3 sftpmover_getpass.py
```

22. Enter the password: `alta3`

23. SSH to bender and confirm that the file `file_to_move.txt` was moved.

```
student@bchd:~/mycode/passwd$ ssh bender@10.10.2.3
```

24. SSH to bender and confirm that your script `file_to_move.txt` was moved.

```
bender@bender:~$ ls
```

25. Exit bender.

```
bender@bender:~$ exit
```

26. **CUSTOMIZATION REQUEST 01 -** Create a function, `movethemfiles()`, that is passed `sftp` as a value. Within this function place the logic found under the comment 'iterate across the files within directory'

27. **CUSTOMIZATION REQUEST 02 -** Collect additional input from the user to determine where on the target host the files should be placed. Ensure the target location exists before moving the files.

28. If you're tracking your code in an SCM platform, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "sftp with username and password"`
    - `git push origin HEAD`

65. Paramiko - SSH with RSA Keys

Lab Objective

The objective of this lab is to learn to push commands across an SSH connection using Paramiko. This lab uses RSA keypairs to connect. However, Paramiko can also work with passwording.

Procedure

1. We'll be using the **planetexpress** team for this lab! Run the command below to prepare your environment.

```
student@bchd:~$ bash ~/px/scripts/full-setup.sh
```

2. Create a directory to work in and move into it.

```
student@bchd:~$ mkdir -p ~/mycode/paramikosshrsa/ && cd ~/mycode/paramikosshrsa/
```

3. For this lab to run we will need Paramiko. Download it from the `pypi` repo with the `pip` tool.

```
student@bchd:~$ python3 -m pip install paramiko
```

4. Create a new script.

```
student@bchd:~/mycode/paramikosshrsa/$ vim ~/mycode/paramikosshrsa/learntossh.py
```

5. Create the following script:

```
#!/usr/bin/python3
"""Alta3 Research | RZFeeser@alta3.com
   Issuing commands across a SSH channel"""

import os

## import paramiko so we can talk SSH
import paramiko

## shortcut issuing commands to remote
def commandissue(command_to_issue, sshsession):
    ssh_stdin, ssh_stdout, ssh_stderr = sshsession.exec_command(command_to_issue)
    return ssh_stdout.read()

def main():
    sshsession = paramiko.SSHClient()

    ############# IF YOU WANT TO CONNECT USING UN/PW ################
    #sshsession.connect(server, username=username, password=password)
    ############## IF USING KEYS ####################################

    ## mykey is our private key
    mykey = paramiko.RSAKey.from_private_key_file("/home/student/.ssh/id_rsa")

    ## if we never went to this SSH host, add the fingerprint to the known host file
    sshsession.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    ## creds to connect
    sshsession.connect(hostname="10.10.2.3", username="bender", pkey=mykey)

    ## a simple list of commands to issue across our connection
    our_commands = ["touch sshworked.txt", "touch create.txt", "touch file3.txt", "ls"]

    ## cycle through our commands and issue them on the far end
    for x in our_commands:
        print(commandissue(x, sshsession))
main()
```

6. Save and exit, then run your script.

```
student@bchd:~/mycode/paramikosshrsa/$ python3 ~/mycode/paramikosshrsa/learntossh.py
```

7. Yuck. The tiny 'b' is appearing because we need to decode the `bytes` of the string. The fix is always the same if you ever see this error. Change the last line of your script to read like the following and it will go away:

```
        print(commandissue(x, sshsession).decode('utf-8'))
```

8. Not bad, but currently our code only cycles across a single user (bender). Let's design a script that loops across several usernames and creates a file in each user's home directory.

```
student@bchd:~/mycode/paramikosshrsa/$ vim ~/mycode/paramikosshrsa/moressh.py
```

9. Create the following script:

```
#!/usr/bin/python3
"""Alta3 Research | rzfeeser@alta3.com
   Learning about Python SSH"""

import paramiko

def main():
    """Our runtime code that calls other functions"""
    # describe the connection data
    credz = [
             {"un": "bender", "ip": "10.10.2.3"},
             {"un": "zoidberg", "ip": "10.10.2.5"},
             {"un": "fry", "ip": "10.10.2.4"}
            ]

    # harvest private key for all 3 servers
    mykey = paramiko.RSAKey.from_private_key_file("/home/student/.ssh/id_rsa")

    # loop across the collection credz
    for cred in credz:
        ## create a session object
        sshsession = paramiko.SSHClient()

        ## add host key policy
        sshsession.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        ## display our connections
        print("Connecting to... " + cred.get("un") + "@" + cred.get("ip"))

        ## make a connection
        sshsession.connect(hostname=cred.get("ip"), username=cred.get("un"), pkey=mykey)

        ## touch the file goodnews.everyone in each user's home directory
        sshsession.exec_command("touch /home/" + cred.get("un") + "/goodnews.everyone")

        ## list the contents of each home directory
        sessin, sessout, sesserr = sshsession.exec_command("ls /home/" + cred.get("un"))

        ## display output
        print(sessout.read().decode('utf-8'))

        ## close/cleanup SSH connection
        sshsession.close()

    print("Thanks for looping with Alta3!")

main()
```

10. Save and exit.

11. Run your new script.

```
student@bchd:~/mycode/paramikosshrsa/$ python3 ~/mycode/paramikosshrsa/moressh.py
```

12. Great job! If you need a further challenge, check out the following suggestions.

13. **CUSTOMIZATION REQUEST 01 -** Read the connection information into your script from an external file (username and IPs).

14. **CUSTOMIZATION REQUEST 02 -** Make the output returned by the `ls` command written out to a file `results.log`. Be sure to provide some kind of explanation as to the host that the results were returned from.

15. If you're tracking your code in and SCM, issue the following commands:

    - `cd ~/mycode`
    - `git add *`
    - `git commit -m "SSH and paramiko"`
    - `git push origin HEAD`