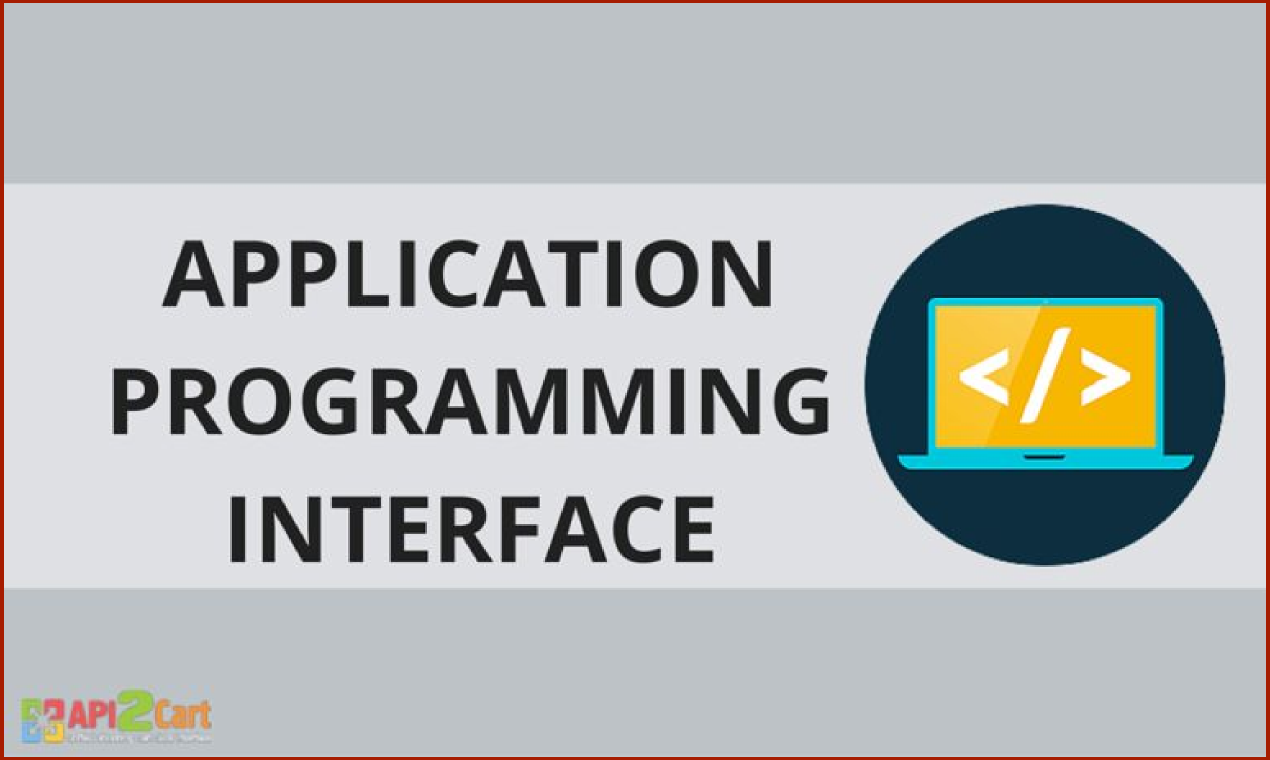
# Enabling Technologies for Data Science

Lesson 5 - APIs



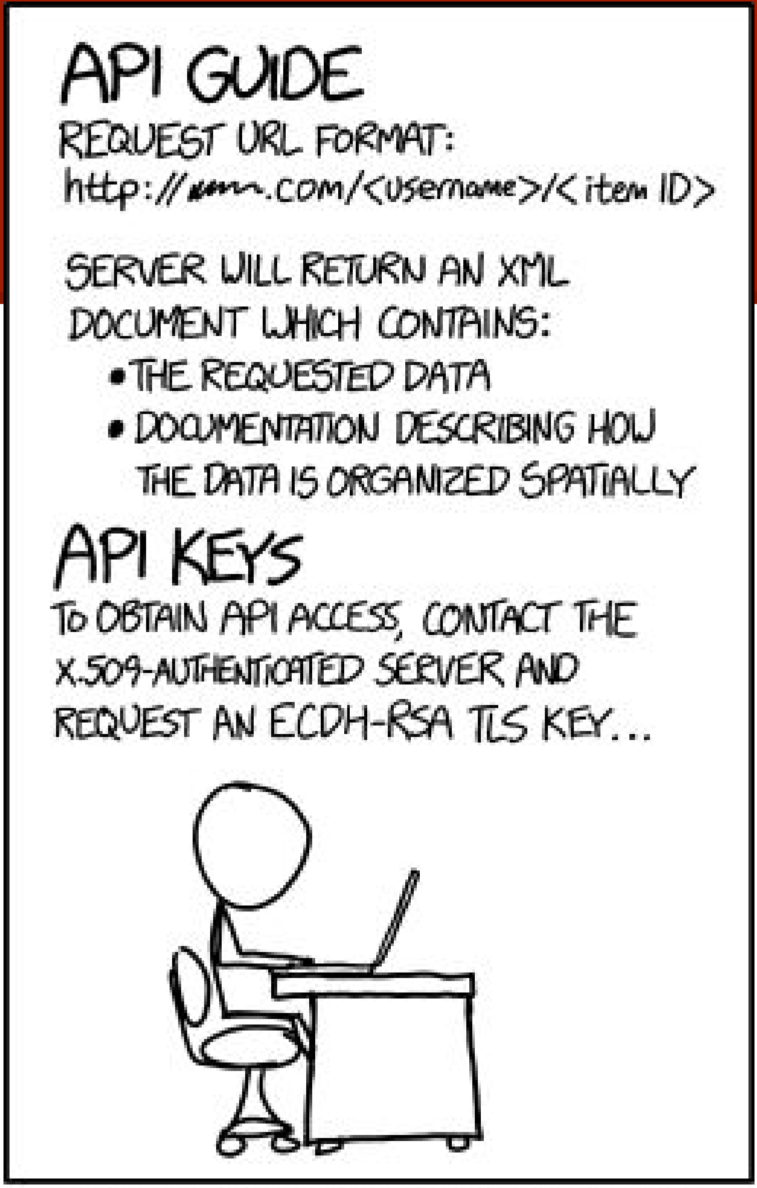
TINKERCADEMY

An Infocomm Club Appointed Vendor

# Intro to APIs

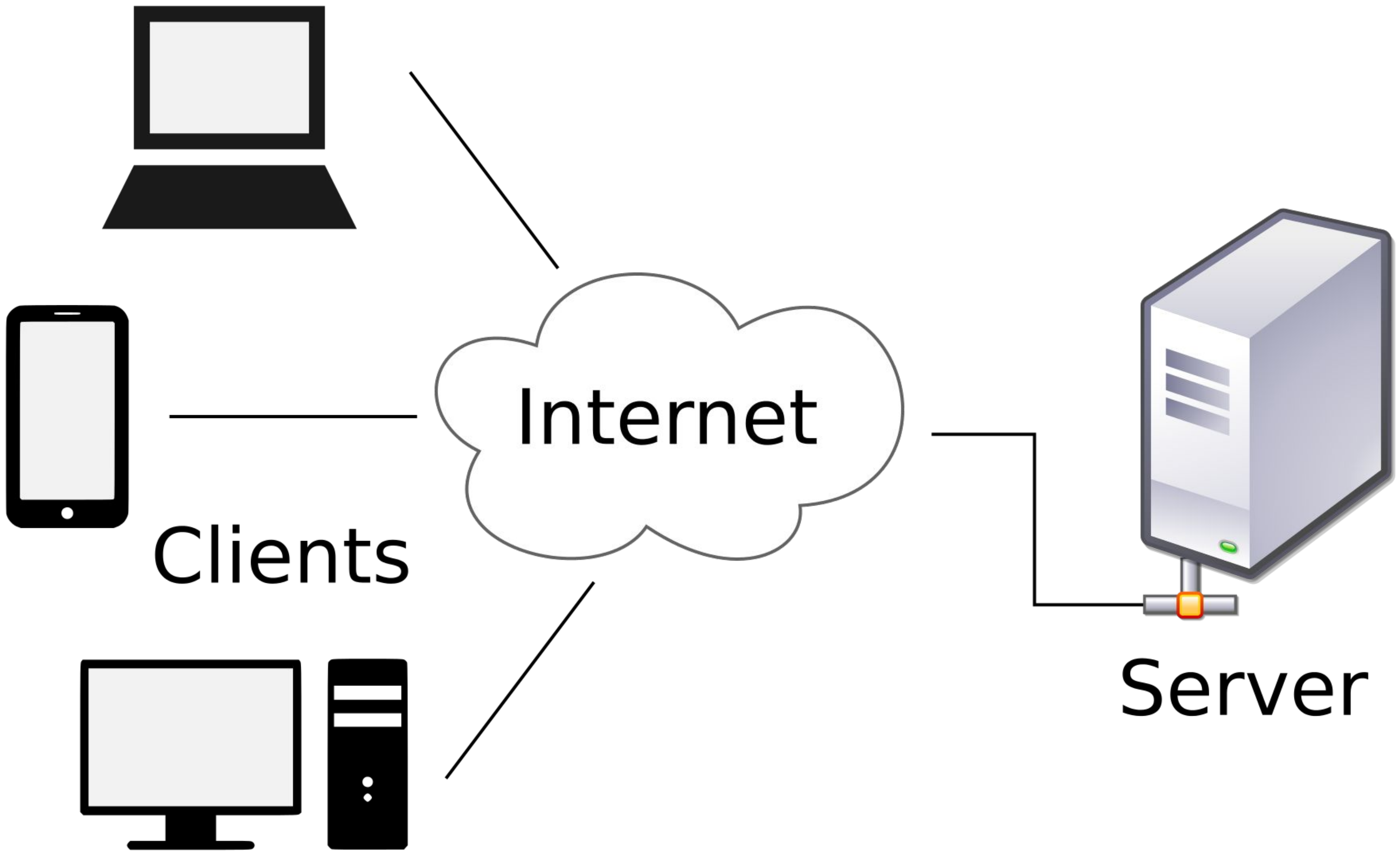
**API**: **A**pplication **P**rogramming **I**nterface

APIs are used by online services as a standardised way of accessing their data.


API GUIDE
REQUEST URL FORMAT:
http://[illegible].com/<username>/<item ID>
SERVER WILL RETURN AN XML DOCUMENT WHICH CONTAINS:
• THE REQUESTED DATA
• DOCUMENTATION DESCRIBING HOW THE DATA IS ORGANIZED SPATIALLY
API KEYS
TO OBTAIN API ACCESS, CONTACT THE X.509-AUTHENTICATED SERVER AND REQUEST AN ECDH-RSA TLS KEY...

IF YOU DO THINGS RIGHT, IT CAN TAKE PEOPLE A WHILE TO REALIZE THAT YOUR "API DOCUMENTATION" IS JUST INSTRUCTIONS FOR HOW TO LOOK AT YOUR WEBSITE.

# Clients and Servers



# Intro to HTTP

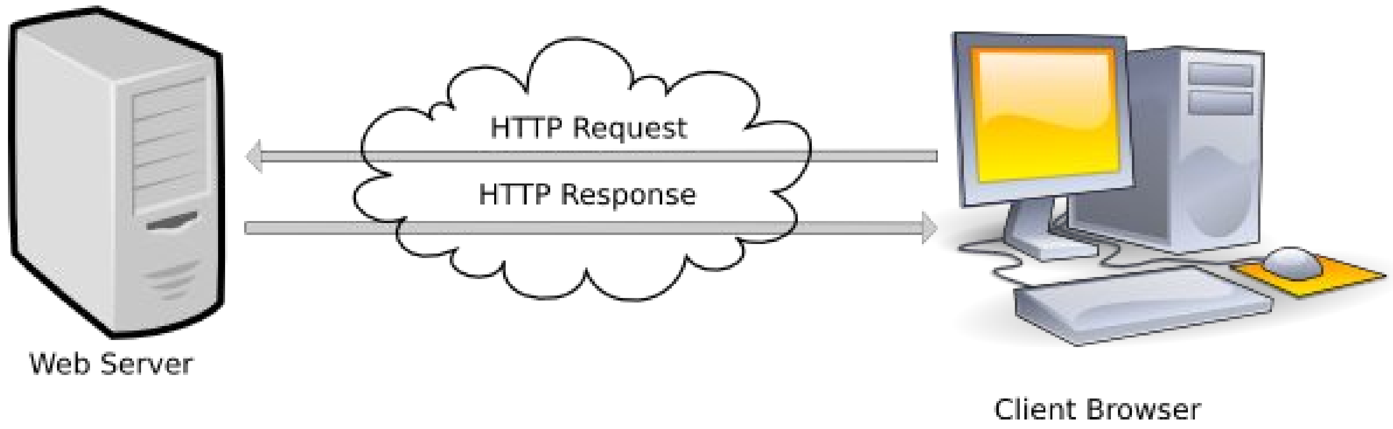
URL = Uniform Resource Locator (every doc has a unique URL)

Schemes:

HTTP = HyperText Transfer Protocol

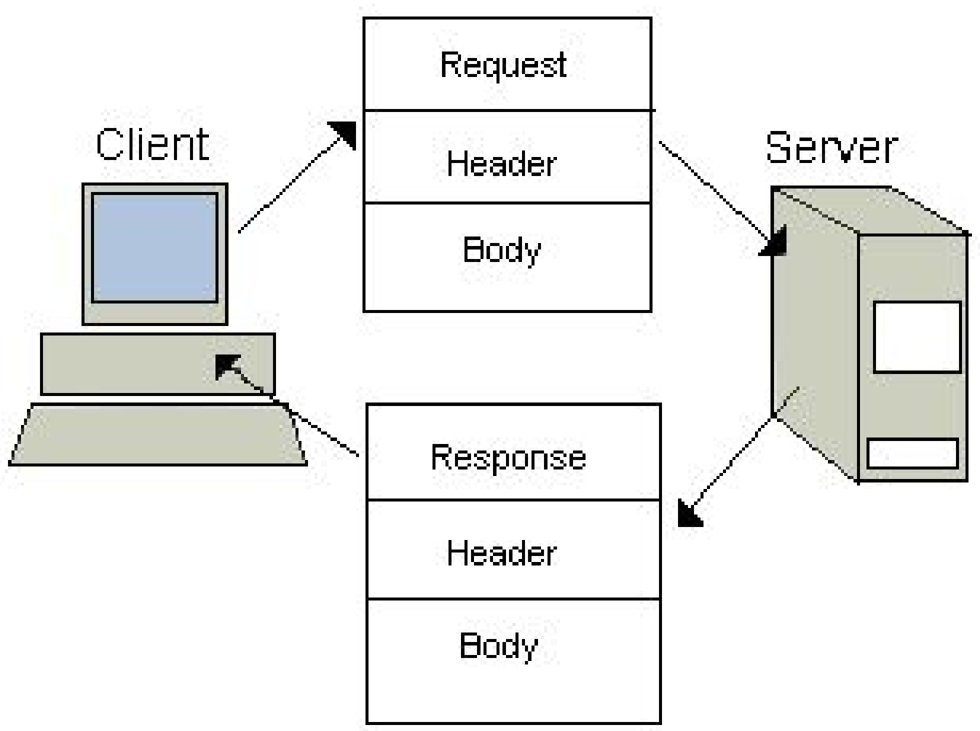
HTTPS = Secure HyperText Transfer Protocol

FTP = File Transfer Protocol



# HTTP Requests/Responses

3 parts to an HTTP request/response:



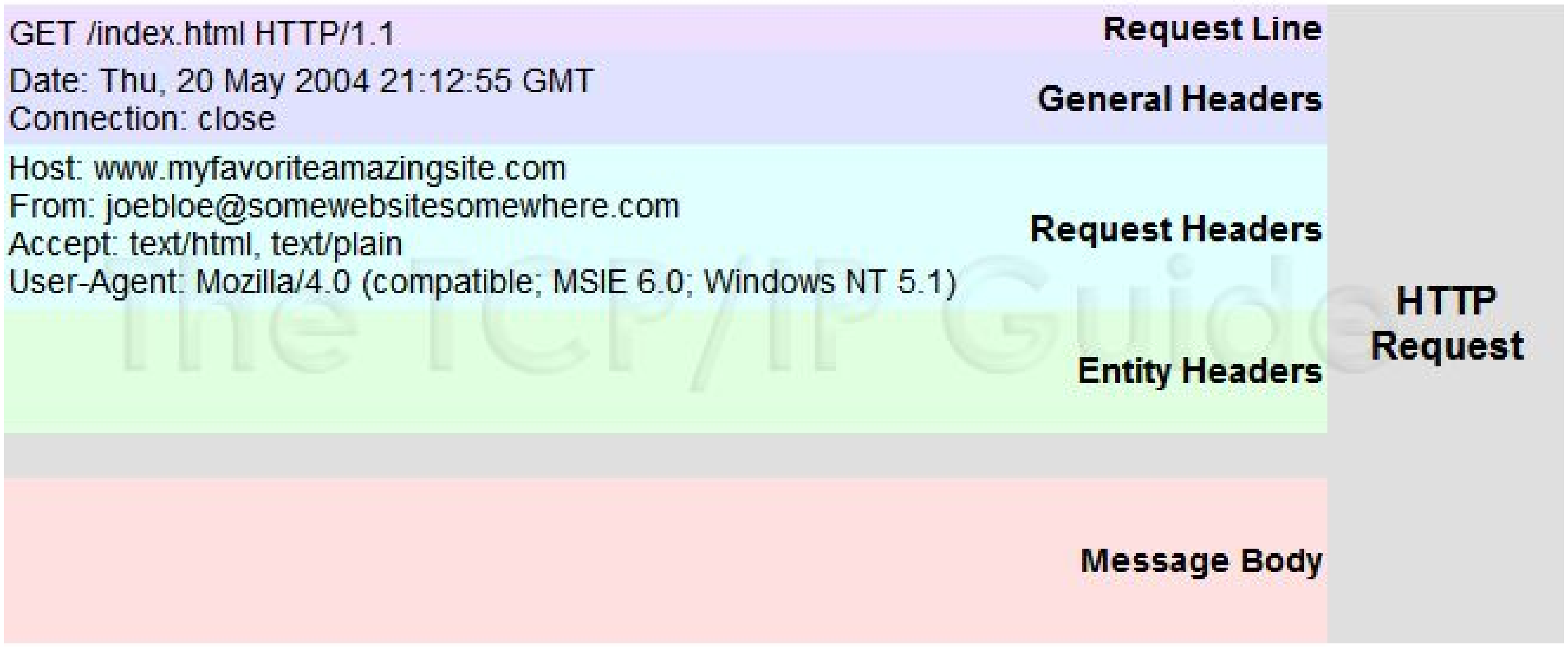
## Request Methods

Commonly:

- GET - retrieves information
- POST - sends information
- PUT - updates information
- DELETE - removes information

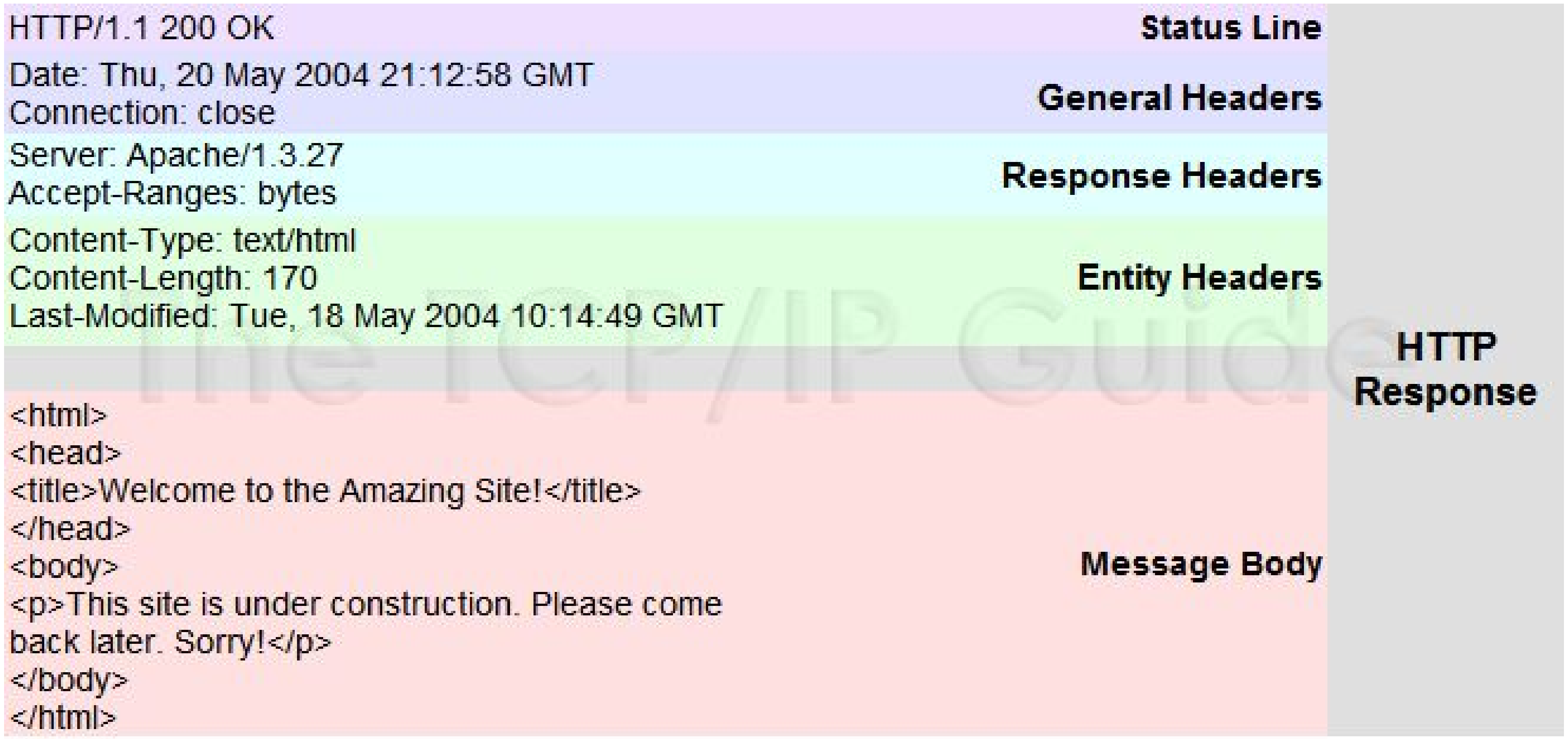
*Also: OPTIONS, HEAD, TRACE, CONNECT*

# Example of an HTTP Request



```
GET /index.html HTTP/1.1
Date: Thu, 20 May 2004 21:12:55 GMT
Connection: close
Host: www.myfavoriteamazingsite.com
From: joebloe@somewebsitesomewhere.com
Accept: text/html, text/plain
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
```

# Example of an HTTP Response



```
HTTP/1.1 200 OK
Date: Thu, 20 May 2004 21:12:58 GMT
Connection: close
Server: Apache/1.3.27
Accept-Ranges: bytes
Content-Type: text/html
Content-Length: 170
Last-Modified: Tue, 18 May 2004 10:14:49 GMT

<html>
<head>
<title>Welcome to the Amazing Site!</title>
</head>
<body>
<p>This site is under construction. Please come
back later. Sorry!</p>
</body>
</html>
```

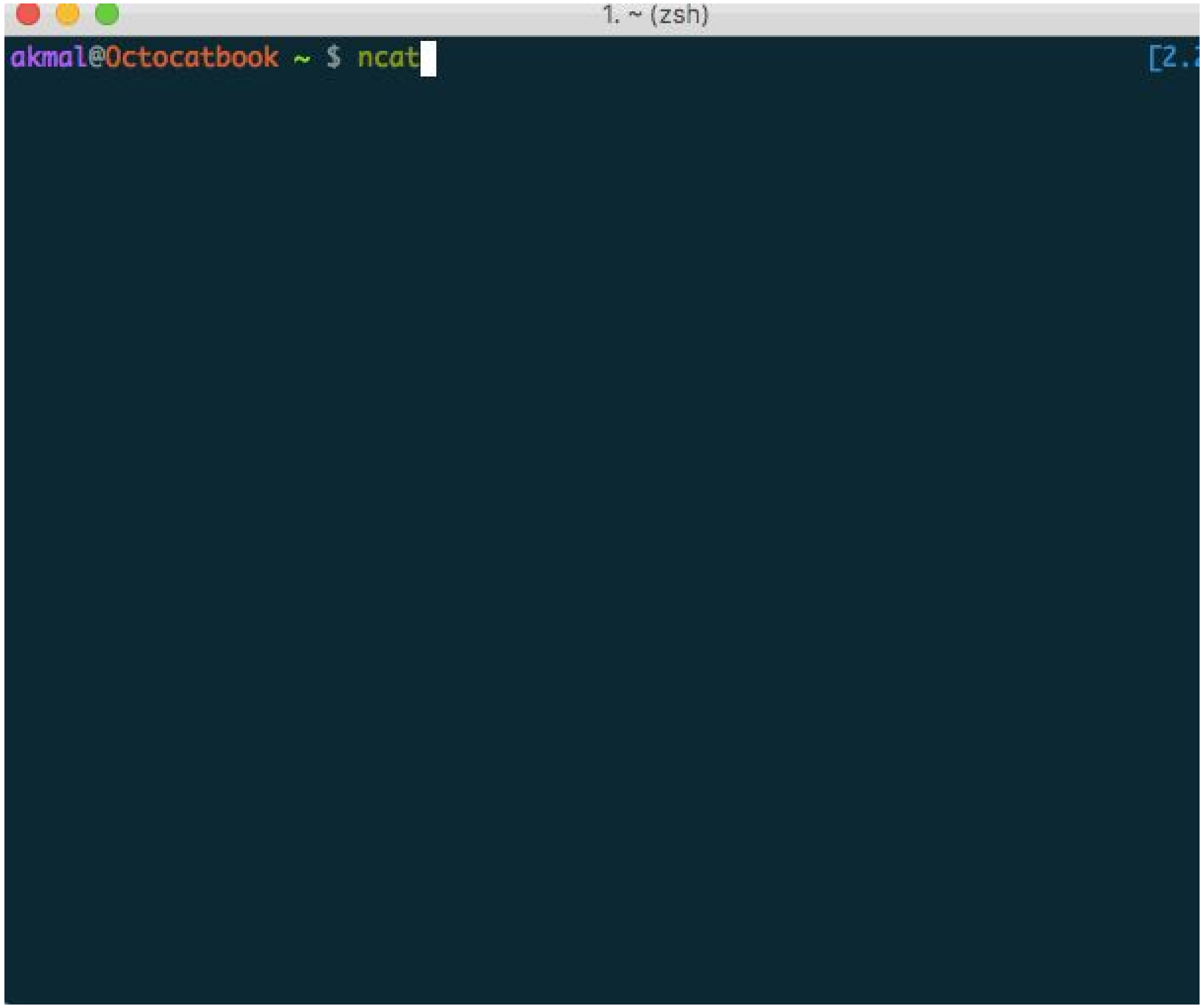
## HTTP Requests

Request Line - specifies the request method (GET, POST, PUT, DELETE, etc.)

Header - contains additional info about the request (e.g. authorisation, content type, cache info, encoding info, etc.)

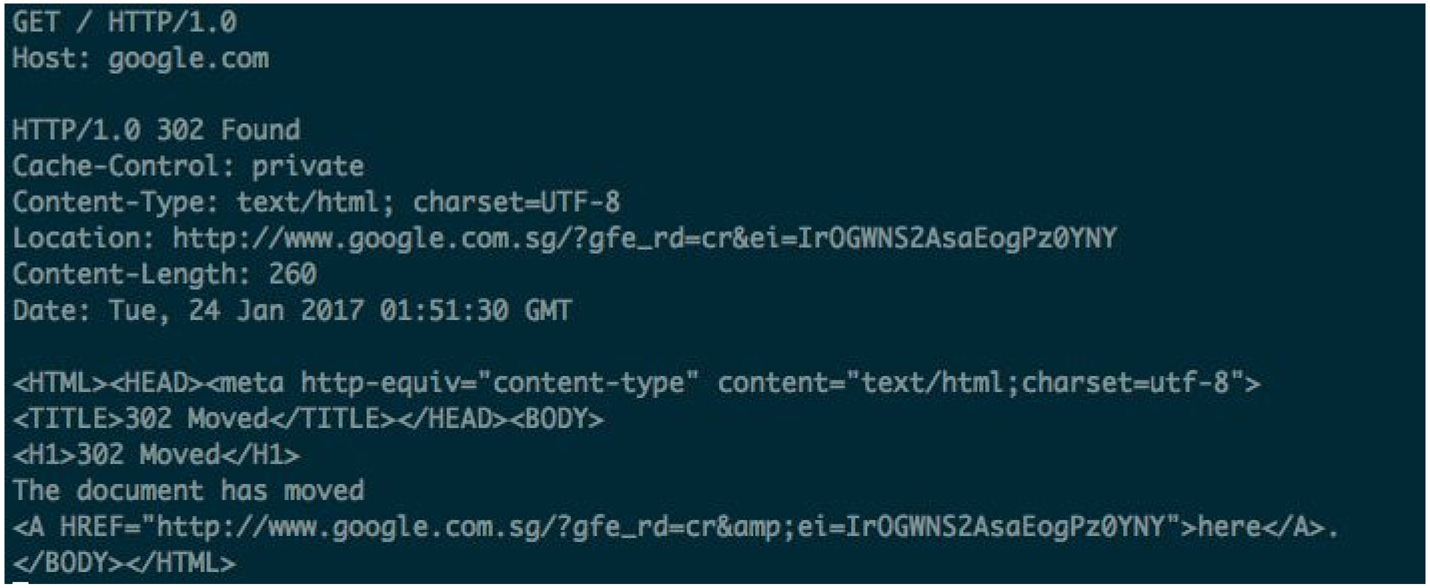
Body - this contains any information (in the case of POST or PUT requests). It might also be empty (in the case of GET requests)

# HTTP request with ncat



```
akmal@Octocatbook ~ $ ncat                                        [2.
```

# HTTP request with google.com

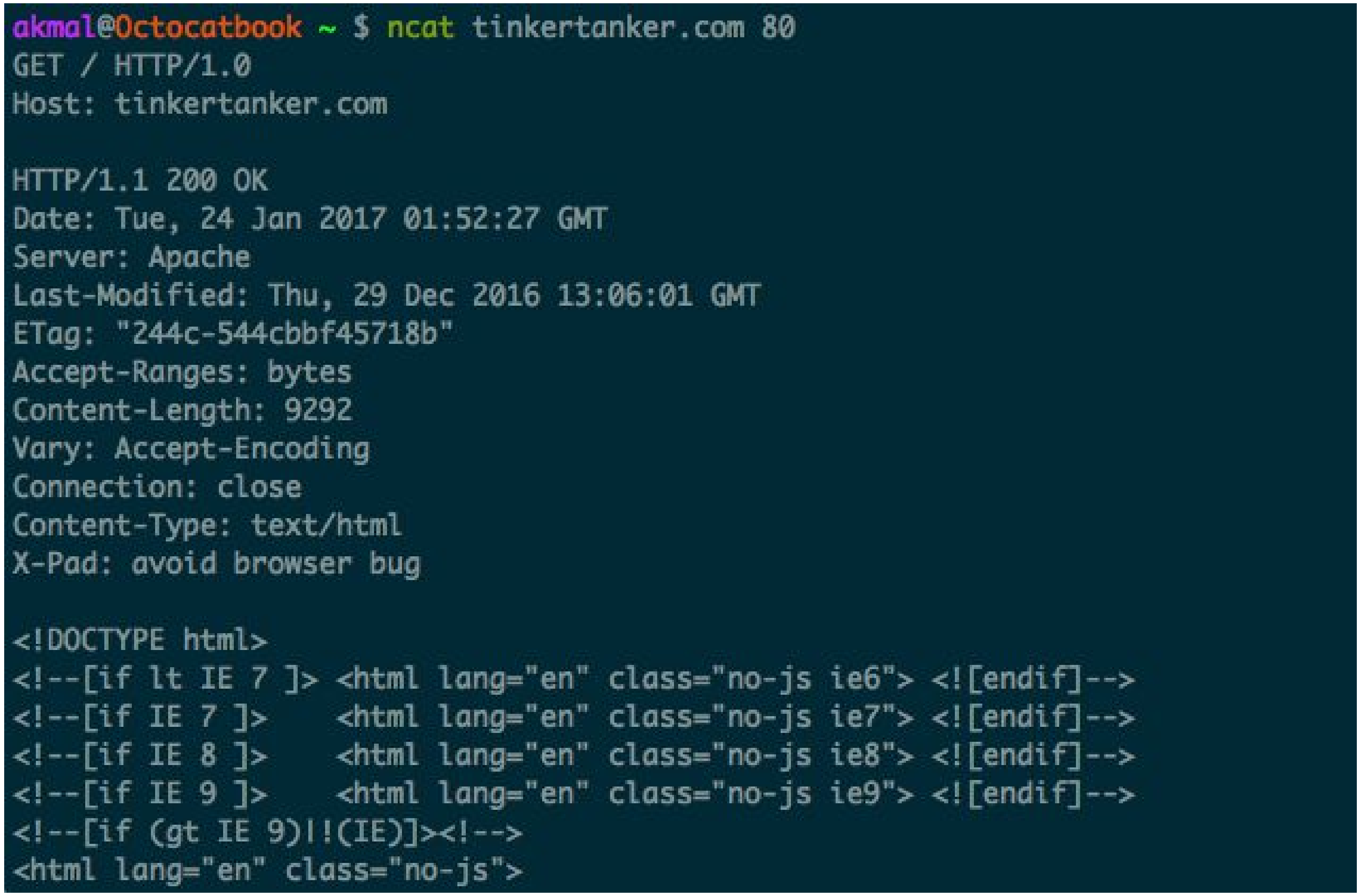


```
GET / HTTP/1.0
Host: google.com

HTTP/1.0 302 Found
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Location: http://www.google.com.sg/?gfe_rd=cr&ei=IrOGWNS2AsaEogPz0YNY
Content-Length: 260
Date: Tue, 24 Jan 2017 01:51:30 GMT

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.com.sg/?gfe_rd=cr&amp;ei=IrOGWNS2AsaEogPz0YNY">here</A>.
</BODY></HTML>
```

## HTTP request with tinkertanker.com

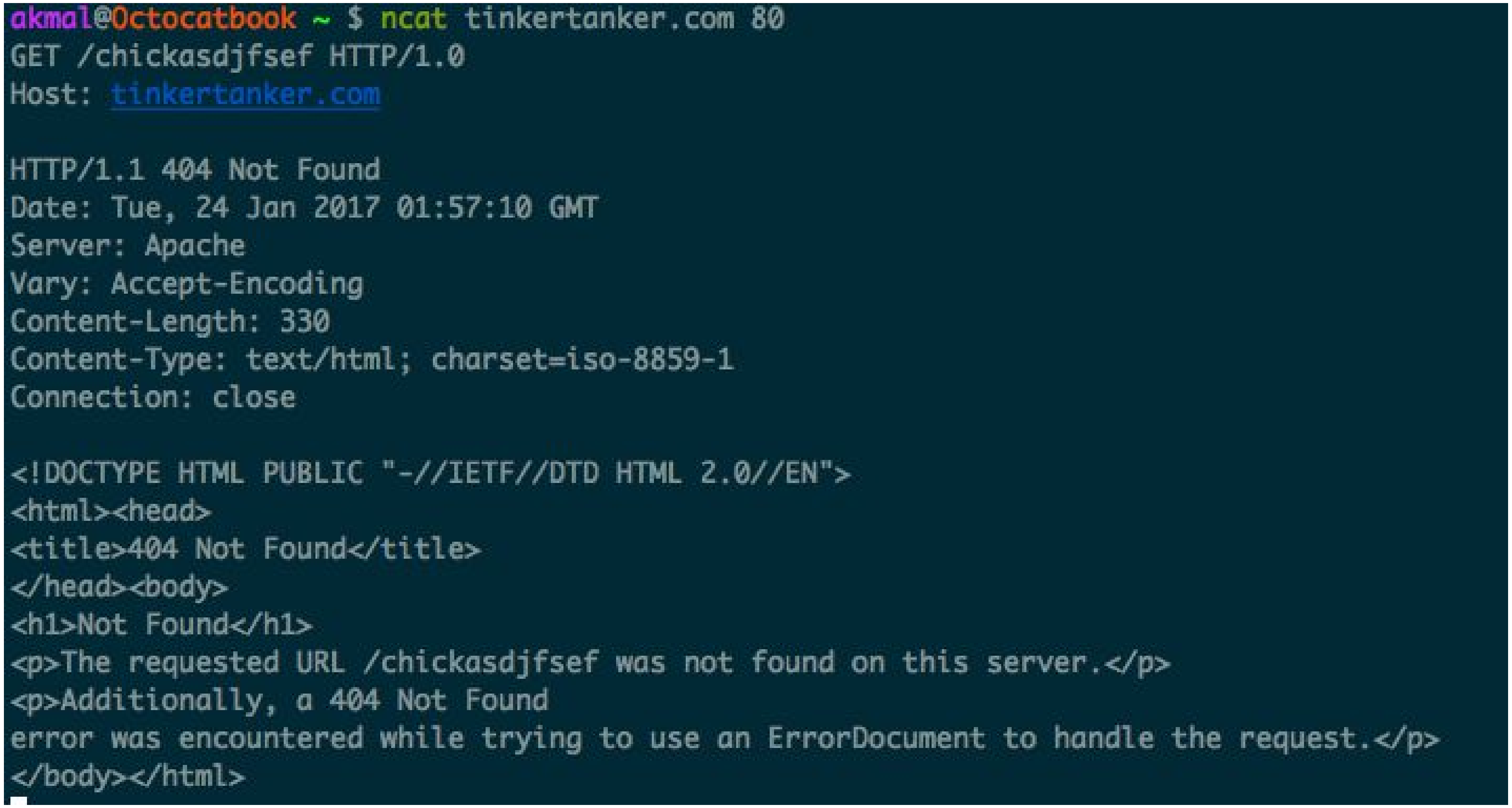


```
akmal@Octocatbook ~ $ ncat tinkertanker.com 80
GET / HTTP/1.0
Host: tinkertanker.com

HTTP/1.1 200 OK
Date: Tue, 24 Jan 2017 01:52:27 GMT
Server: Apache
Last-Modified: Thu, 29 Dec 2016 13:06:01 GMT
ETag: "244c-544cbbf45718b"
Accept-Ranges: bytes
Content-Length: 9292
Vary: Accept-Encoding
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

<!DOCTYPE html>
<!--[if lt IE 7 ]> <html lang="en" class="no-js ie6"> <![endif]-->
<!--[if IE 7 ]>    <html lang="en" class="no-js ie7"> <![endif]-->
<!--[if IE 8 ]>    <html lang="en" class="no-js ie8"> <![endif]-->
<!--[if IE 9 ]>    <html lang="en" class="no-js ie9"> <![endif]-->
<!--[if (gt IE 9)|!(IE)]><!-->
<html lang="en" class="no-js">
```

# HTTP request for nonsense page



```
akmal@Octocatbook ~ $ ncat tinkertanker.com 80
GET /chickasdjfsef HTTP/1.0
Host: tinkertanker.com

HTTP/1.1 404 Not Found
Date: Tue, 24 Jan 2017 01:57:10 GMT
Server: Apache
Vary: Accept-Encoding
Content-Length: 330
Content-Type: text/html; charset=iso-8859-1
Connection: close

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /chickasdjfsef was not found on this server.</p>
<p>Additionally, a 404 Not Found
error was encountered while trying to use an ErrorDocument to handle the request.</p>
</body></html>
```

## Response Codes

## Response Codes

Examples of GET Response Codes:

- 200 – everything went okay
- 301 – the server is redirecting you
- 400 – the server thinks you made a bad request
- 401 – the server thinks you're not authenticated
- 403 – you lack the permissions to access the resource
- 404 – the resource you tried to access wasn't found

## Response Formats

JSON = JavaScript Object Notation

XML = eXtensible Markup Language

Most APIs will give you a choice of the format in which you would like to receive your response. We'll be using python to handle both of these response types.

# JSON

JSON is written in name:value pairs

```
Ex: {"name":"Squirtle", "type":"water"}
```

## JSON Example

```
{"breakfast-menu":[
        {"name":"Belgian Waffles", "price":"$5.95", "calories":"650", "desc":"Our
famous Belgian Waffles with plenty of real maple syrup."},
        {"name":"French Toast", "price":"$4.50", "calories":"600", "desc":"Thick
slices made from our homemade sourdough bread."},
        {"name":"Homestyle Breakfast", "price":"$6.95", "calories":"950",
"desc":"Two eggs, bacon or sausage, toast, and our ever-popular hash
browns."},
]}
```

# XML

XML is very similar to HTML; it contains data inside tags.

```
Ex:
<pokemon>
  <name>Squirtle</name>
  <type>Water</type>
</pokemon>
```

# XML Example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<breakfast_menu>
<food>
        <name>Belgian Waffles</name>
        <price>$5.95</price>
        <description>Our famous Belgian Waffles with plenty of real maple syrup</description>
        <calories>650</calories>
</food>
<food>
        <name>French Toast</name>
        <price>$4.50</price>
        <description>Thick slices made from our homemade sourdough bread</description>
        <calories>600</calories>
</food>
<food>
        <name>Homestyle Breakfast</name>
        <price>$6.95</price>
        <description>Two eggs, bacon or sausage, toast, and our ever-popular hash browns</description>
        <calories>950</calories>
</food>
</breakfast_menu>
```

## JSON/XML

We’ll learn how to handle these response formats in python shortly, but first let’s try a real world example…

**data.gov.sg**

https://developers.data.gov.sg/environment/pm25

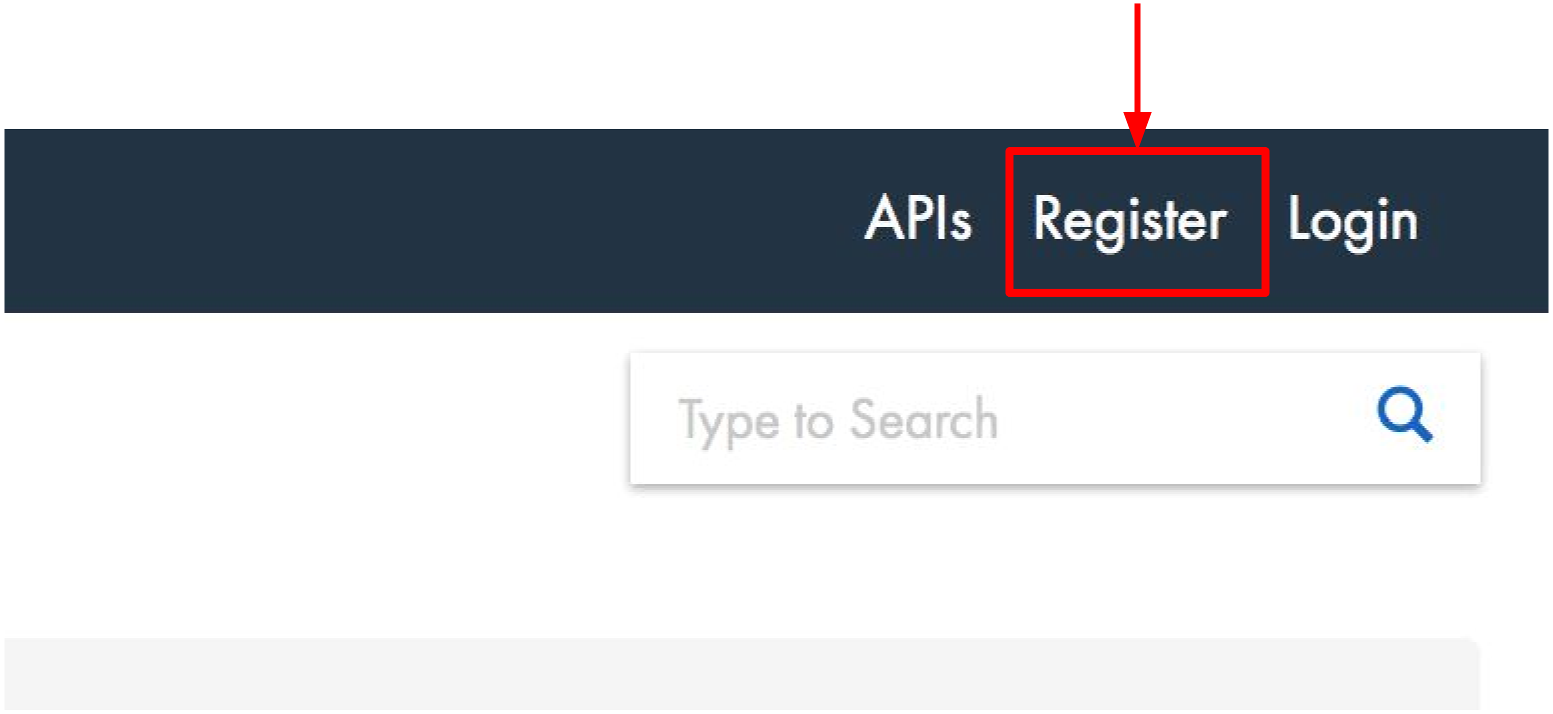
## data.gov.sg API

Every site will have different rules to follow regarding use of their API.

Most APIs require a key. This allows sites to track requests to prevent abuse.

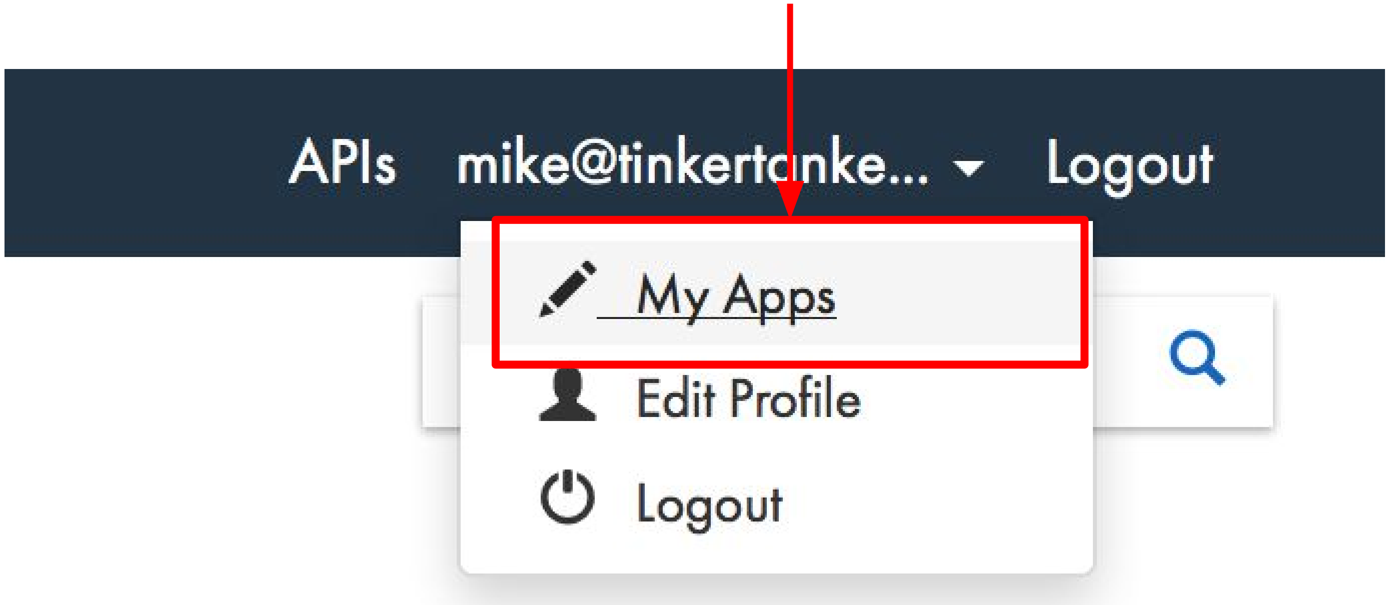
# data.gov.sg API Consumer Key

1. Register for an account on data.gov.sg



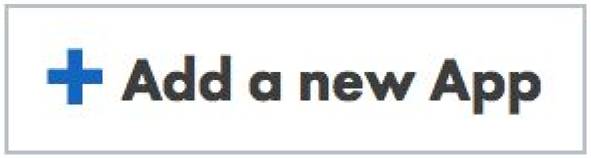
# data.gov.sg API Consumer Key

2. Click ‘My Apps’



# data.gov.sg API Consumer Key

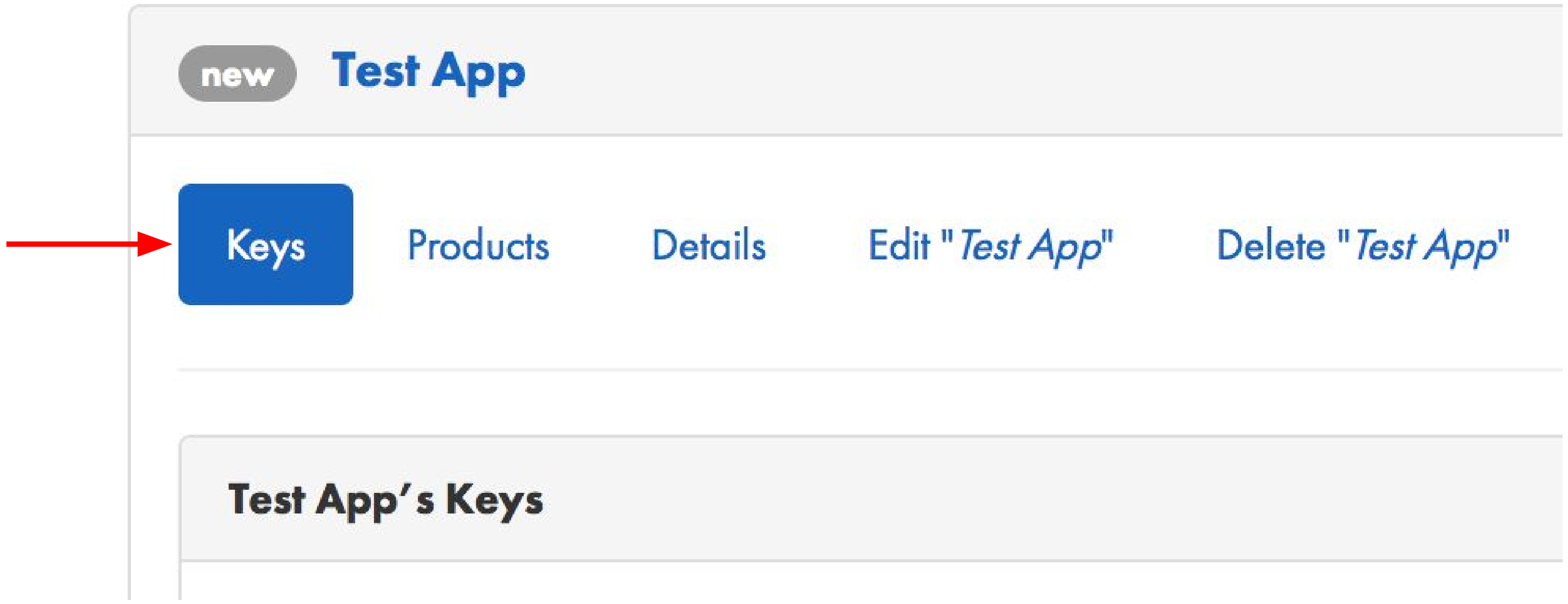
3. Click ‘Add a new App’



4. Name the app anything you want

# data.gov.sg API Consumer Key

5. You will see your consumer key under the ‘Keys’ tab



## data.gov.sg API Consumer Key

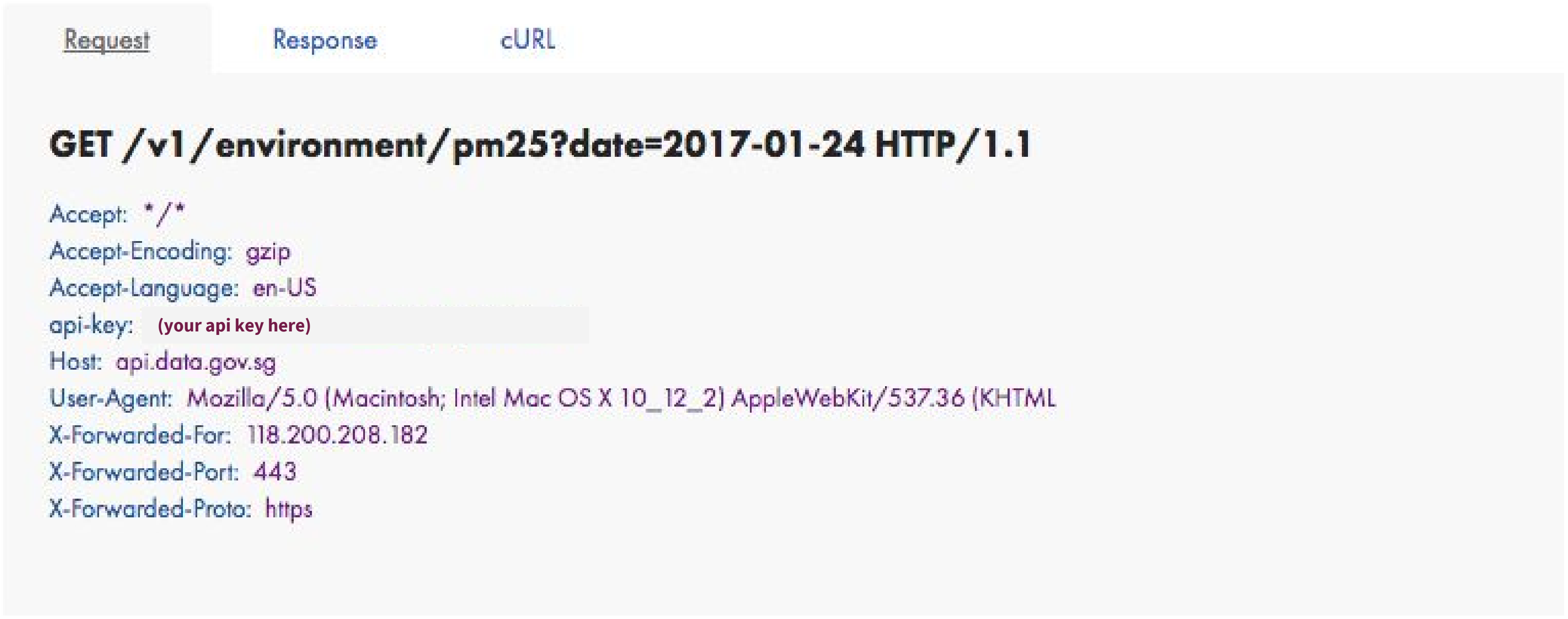
However…

The current web form to request data doesn’t care about authentication keys as long as you supply a key (no matter what the key is)...

But if you want  to access the data via python or command line, you need a valid api-key.

# data.gov.sg API Example

Example request:



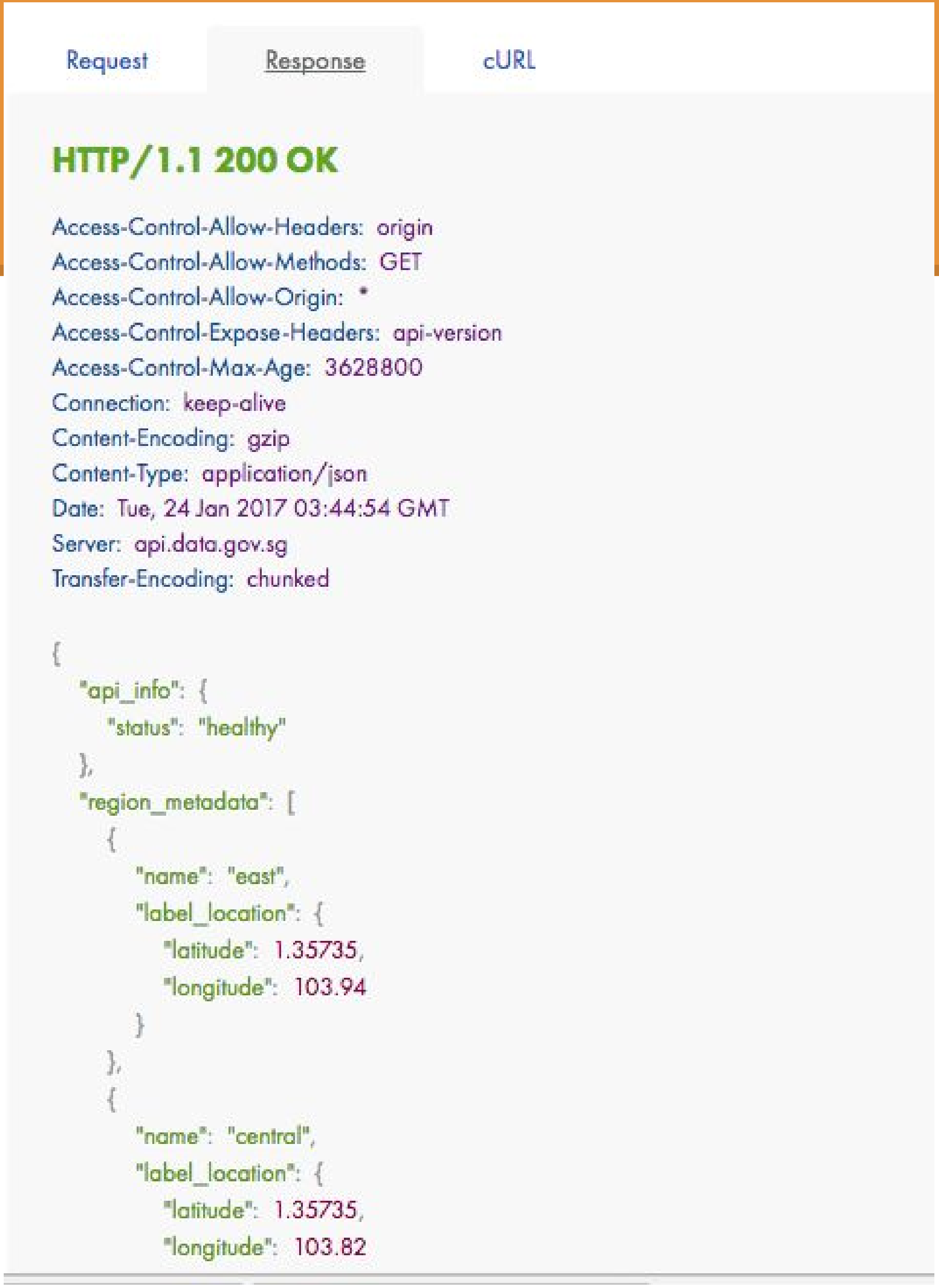
Request Response cURL

```
GET /v1/environment/pm25?date=2017-01-24 HTTP/1.1

Accept: */*
Accept-Encoding: gzip
Accept-Language: en-US
api-key: (your api key here)
Host: api.data.gov.sg
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_2) AppleWebKit/537.36 (KHTML
X-Forwarded-For: 118.200.208.182
X-Forwarded-Port: 443
X-Forwarded-Proto: https
```

# data.gov.sg API Example

Example response:



Request Response cURL

```
HTTP/1.1 200 OK

Access-Control-Allow-Headers: origin
Access-Control-Allow-Methods: GET
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: api-version
Access-Control-Max-Age: 3628800
Connection: keep-alive
Content-Encoding: gzip
Content-Type: application/json
Date: Tue, 24 Jan 2017 03:44:54 GMT
Server: api.data.gov.sg
Transfer-Encoding: chunked

{
  "api_info": {
    "status": "healthy"
  },
  "region_metadata": [
    {
      "name": "east",
      "label_location": {
        "latitude": 1.35735,
        "longitude": 103.94
      }
    },
    {
      "name": "central",
      "label_location": {
        "latitude": 1.35735,
        "longitude": 103.82
```

## Handling JSON in Python

**Try it out:**

Let’s say we want all the PM25 data on 24 Jan, 2017. How woulc
we structure our request url?

# Handling JSON in Python

**Try it out:**

Let's say we want all the PM25 data on 24 Jan, 2017. How woulc
we structure our request url?

**Answer:**

```
https://api.data.gov.sg/v1/environment/pm25?date=2017-01-24
```

## Handling JSON in Python

```
# Import the modules
import urllib.request
import json

# Create a request
url = "https://api.data.gov.sg/v1/environment/pm25?date=2017-01-24"
request = urllib.request.Request(url)

# request is an object with various methods/attributes
request.add_header('api-key', '[YOUR-KEY-HERE]')
```

## Handling JSON in Python

```
# Get the feed and store it
response = urllib.request.urlopen(request)

# response is also an object with various methods/attributes
data = response.read()

# Convert it to a Python dictionary
parsed_json = json.loads(data.decode('utf-8'))

# Format and output
print(json.dumps(parsed_json, indent=2))
```

# Handling JSON in Python

A closer look:

```
# Get the feed and store it
response = urllib.request.urlopen(request)
data = response.read()
```

Here we call `read()` on **response** and we get **data**, a bytes object.

This is where the entire HTTP request and response cycle lives. When the server replies with a response, we store it in **response**, which is a file-like object that we can `read()`.

## Handling JSON in Python

A closer look:

```
# Convert it to a Python dictionary
parsed_json = json.loads(data.decode('utf-8'))
```

Bytes objects can contain anything (text, images, sounds). So, we call `decode()` on it to specify that `data` is a string with utf-8 encoding. This returns a string which happens to be JSON-formatted. Then, we can call `json.loads()` to convert this string into a Python dictionary, and we store that dictionary in `parsed_json`.

## Handling JSON in Python

**To recap:**

1. We create a request and send it to the server, making sure to add appropriate headers (api-key, etc.)
2. We receive back a bytes object (stored in our data variable. This could be anything (text, images, sounds) but in our case, it’s JSON text)
3. We decode the bytes object and convert it into a python dictionary.

# data.gov.sg API

**Try it out:**

Can you find the **average** of the PM25 hourly readings in the **South** on **Jan 24, 2017** by manipulating the JSON response in Python?

Reminder - how to convert an HTTPResponse to a python dictionary:

```
response = urllib.request.urlopen(request)
parsed_json = json.loads(response.read().decode('utf-8'))
```

# data.gov.sg API

**Answer:**

```
...
parsed_json = json.loads(data.decode('utf-8'))

# Add PM2.5 readings over the day and divide to get average
avg = 0.0
for item in parsed_json["items"]:
    avg += item["readings"]["pm25_one_hourly"]["south"]
avg /= len(parsed_json["items"])
print(avg)
```

# data.gov.sg API

## You can also do this (but don’t):

```
import urllib.request, json
print( sum( item["readings"]["pm25_one_hourly"]["south"] for
item in
json.loads(urllib.request.urlopen(urllib.request.Request("https:
//api.data.gov.sg/v1/environment/pm25?date=2017-01-24", headers
= {"api-key":
"BS2mysYRPHmA52SUlZW9G9kHdDKN3rBS"})).read().decode('utf-8'))["i
tems"] ) / 24)
```

## Reddit API

Almost all APIs on the web will have some differences, so we won’t be able to just copy-paste code sadly. Let’s try using the Reddit API.

Reddit is a social news aggregation and discussion website where users can vote up or down posts. It has various subsites (called subreddits) for specific news topics.

Reddit has it’s own API. And it also has PRAW, a Python Reddit API Wrapper, for easy access to the Reddit API.

## Reddit API Exercise

**Try it out:**

Count the number of occurrences of the following words in the top 100 post titles of all time on the pokemonGO subreddit (r/pokemongo/).

a) pokemon
b) pokémon
c) valor
d) mystic
e) instinct
f) dragonite
g) niantic
h) professor oak

## Reddit API Exercise

**Try it out:**

Count the number of occurrences of the following words in the top 100 post titles of all time on the pokemonGO subreddit (r/pokemongo/).

Hint: I recommend using PRAW as it's easier but it's not required! There are also lots of considerations to take into account (uppercase words vs lowercase, etc.).

Hint: You can install the praw module using pip and read up on it at [praw.readthedocs.io](praw.readthedocs.io)

## Uploading Data

So far we’ve just learned how to access data already online. But what if we want to upload and store our own data online?

When might we want to store data online?

Your weather station in the backyard.

A classroom of kids working on science experiments.

A community concerned with pollution.

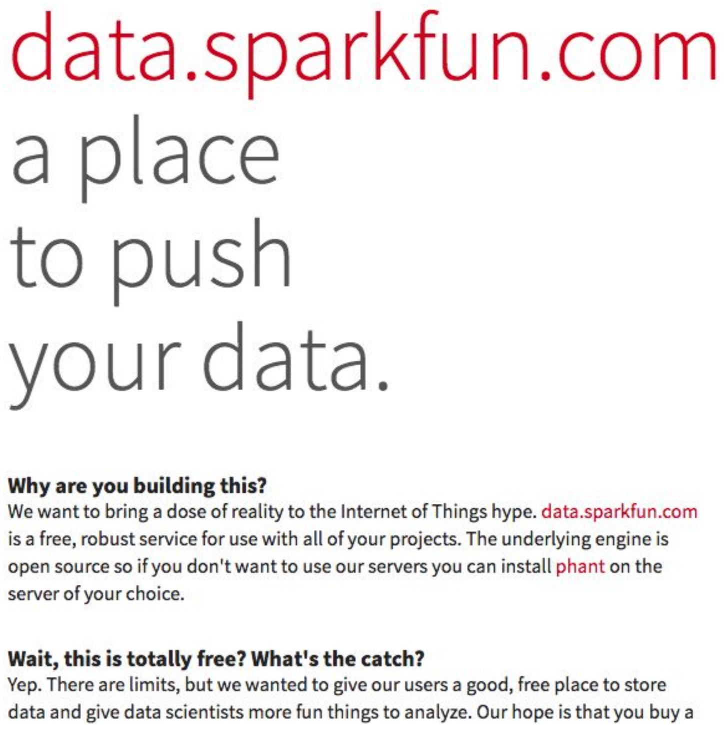
The weight of your dog’s/cat's food dish.

# data.sparkfun.com

We used to use SparkFun's data.sparkfun.com service.

It's powered by **Phant**, a data logging tool designed for collecting IoT data.

But they're taking a break from providing the service :(



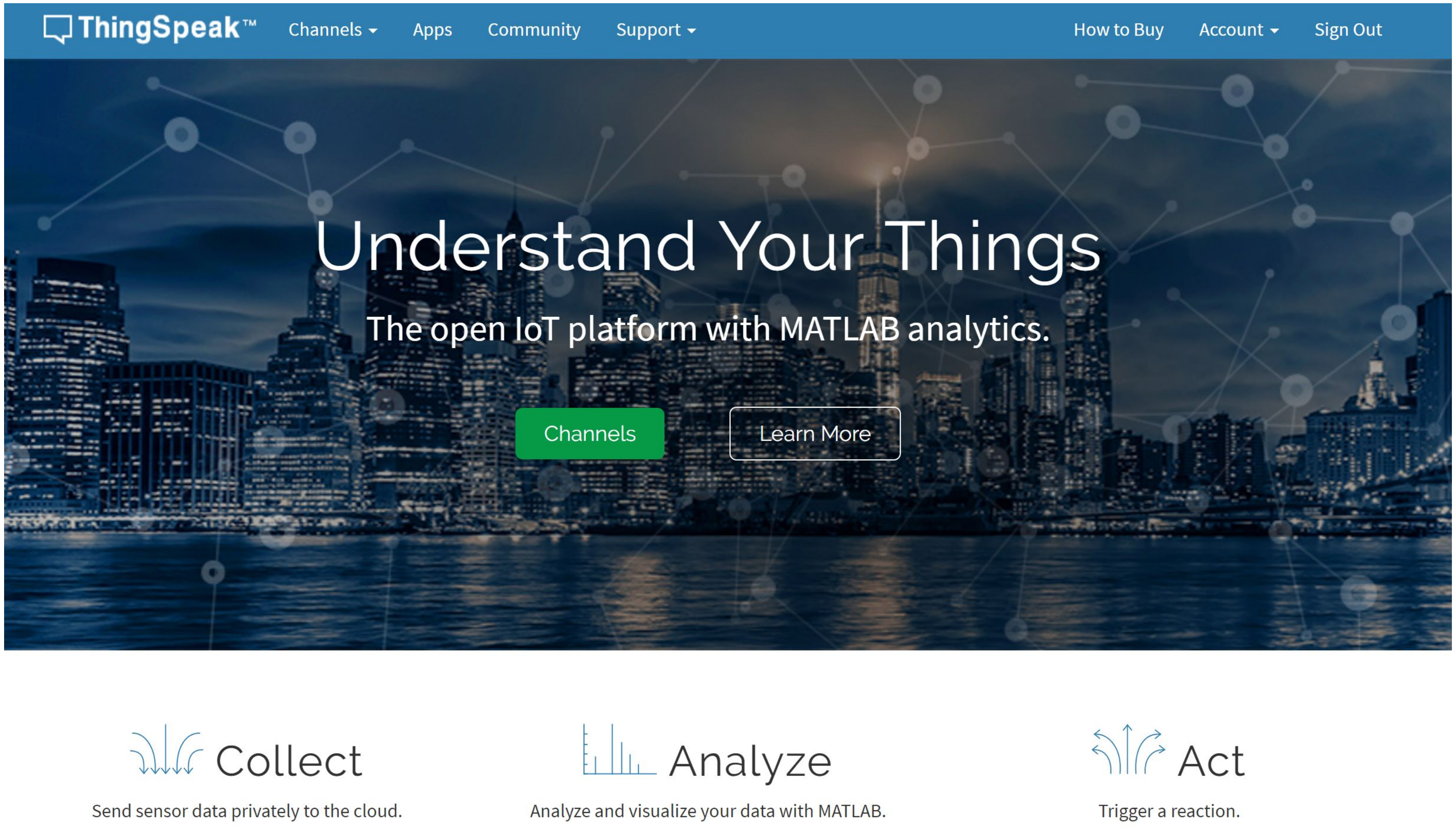
data.sparkfun.com
a place
to push
your data.

**Why are you building this?**
We want to bring a dose of reality to the Internet of Things hype. data.sparkfun.com is a free, robust service for use with all of your projects. The underlying engine is open source so if you don't want to use our servers you can install phant on the server of your choice.

**Wait, this is totally free? What's the catch?**
Yep. There are limits, but we wanted to give our users a good, free place to store data and give data scientists more fun things to analyze. Our hope is that you buy a

# Thingspeak

Instead, we'll use Thingspeak



Understand Your Things

The open IoT platform with MATLAB analytics.

Collect

Send sensor data privately to the cloud.

Analyze

Analyze and visualize your data with MATLAB.

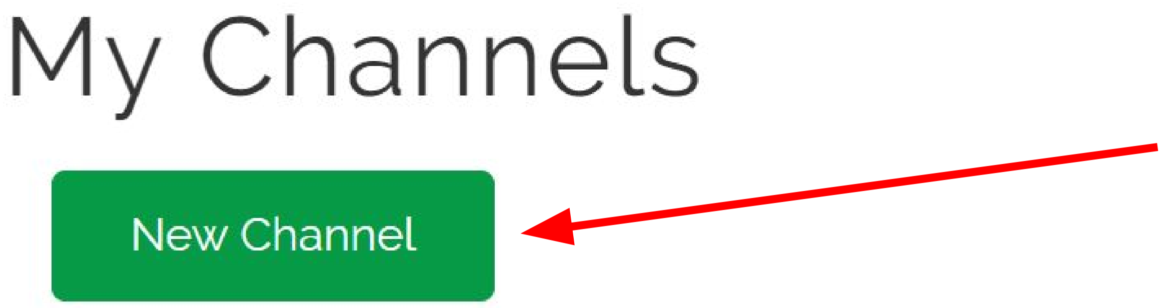
Act

Trigger a reaction.

# Thingspeak

1. First, create an account on Thingspeak

2. Once you're logged in, create a channel



I created a channel with two fields: temperature and humidity. You can use different fields if you want.

# Thingspeak

3. Next, we need to learn how to upload data to our channel. Let's start with a very basic example. We'll create one row of fake data

```
temp = 25
humidity = 86
```

If we can successfully upload one row of data, it should be easy to extend to multiple rows of data.

## Thingspeak

```
#PART 1
import http.client, urllib
key = 'W4BWAIEOZL6EIS7U'  #the key to your thingspeak channel
temp = 25
humidity = 86

data = {'field1': temp, 'field2': humidity, 'key':key }
encodeddata = urllib.parse.urlencode(data)

headers = {}
conn = http.client.HTTPConnection("api.thingspeak.com:80")
```

This is often where username/pass are passed

Some APIs require content_length to be passed as well

# Thingspeak

```
#PART 1
```

```
import http.client, urllib
key = 'W4BWAIEOZL6EIS7U'  #the key to your thingspeak channel
temp = 25
humidity = 86
```

**Here we import modules, set our api-key, and get our data (hardcoded in this case)**

```
data = {'field1': temp, 'field2': humidity, 'key':key }
encodeddata = urllib.parse.urlencode(data)
```

**Here we load our data into a dictionary and encode it**

```
headers = {} #this is often where username/pass are passed
conn = http.client.HTTPConnection("api.thingspeak.com:80")
```

**Here we setup the headers (blank in this example) and setup the connection info**

## Thingspeak

```
#PART 2

try:
    conn.request("POST", "/update", encodeddata, headers)
    response = conn.getresponse()
    print(response.status, response.reason)
    data = response.read()
    conn.close()
except:
    print("connection failed")
```

## Thingspeak

#PART 2

```
try:
    conn.request("POST", "/update", encodeddata, headers)
    response = conn.getresponse()
    print(response.status, response.reason)
    data = response.read()
    conn.close()
except:
    print("connection failed")
```

**Here we connect and send the POST request. It's very good practice to print out the response.status and response.reason for verification that your data was uploaded successfully.**

## Thingspeak

**How to upload data to a stream at data.sparkfun.com:**

1. Import libraries

```
import http.client, urllib
```

2. Create a data dictionary and fill it with data and your key

```
data = {}
```

3. Encode the data to prep it for sending using:

```
urllib.parse.urlencode(data)
```

4. Create headers and fill them with our info

```
headers = {}
```

# Thingspeak

5. Initiate connection

```
conn = http.client.HTTPConnection("[YOUR-SERVER-URL]")
```

6. Make a POST request

```
conn.request(requesttype, "/input/[PUBLIC-KEY].txt",
    encodeddata, headers)
```

7. Get the server’s response and print it for feedback

```
response = conn.getresponse()
print(response.status, response.reason)
```

If you have more than one line of data to upload, go back to step 2 and repeat until all your data is uploaded

# Headers

Note that your headers set can have lots of entries. Some might be required, some might be optional - it varies depending on the website:

```
headers["Content-Type"] = "application/x-www-form-urlencoded"
headers["Connection"] = "close" #required for data.sparkfun
headers["Content-Length"] =  #length of data
headers["Phant-Private-Key"] =  #private key for data.sparkfun
```

## Uploading Data

**Try it out:**

Imagine we want to measure the weight of Grumpy Cat’s food bowl every hour to see if he still has enough food leftover. Create your own Thingspeak stream and upload the data (sent via Slack):

## Uploading Data

My channel: **https://thingspeak.com/channels/308226**

**Note:** Timestamps are added automatically. (This isn’t that useful in our example, but it is very useful if we’re logging data automatically (e.g. psi, weather, etc.) as we won’t have to worry about uploading a time field.)