**Making a Request**

You saw a request in the first exercise. Now it's time for you to make your own! (Don't worry, we'll help.)

On [line 1](javascript:void(0)), we've importedurlopen from the urllib2**module**, which is the Python way of bringing in additional functionality we'll need to make our HTTP request. A module is just a collection of extra Python tools.

On [line 4](javascript:void(0)), we'll use urlopen on placekitten.com in preparation for our GET request, which we make when we read from the site on [line 5](javascript:void(0)). (On [line 6](javascript:void(0)), we limit the response to specific character numbers to control the input we get back—this is what gives us our cat image.)

**The Four Verbs**

The number of HTTP methods you'll use is quite small—there are just four HTTP "verbs" you'll need to know! They are:

1. GET: retrieves information from the specified source.
2. POST: sends new information to the specified source.
3. PUT: updates existing information of the specified source.
4. DELETE: removes existing information from the specified source.

So when we sent our GET request to placekitten.com usingurlopen(), we retrieved information. When you add to or update your blog, you're sending POST or PUT requests; when you delete a tweet, there goes a DELETE request.

**Anatomy of a Request**

An HTTP request is made up of three parts:

1. The **request line**, which tells the server what kind of request is being sent (GET, POST, etc.) and what resource it's looking for;
2. The **header**, which sends the server additional information (such as which client is making the request)
3. The **body**, which can be empty (as in a GET request) or contain data (if you're POSTing or PUTing information, that information is contained here).

**Making a POST Request**

1. Using the Requests library, you can make a POST request by using the requests.post()method. You aren't just **GET**ting data with a POST - you can pass your own data into the request as well, like so:

requests.post("http://placekitten.com/", data="myDataToPost")

We're going to make the same request as the one shown on [line 2](javascript:void(0)) through [line 5](javascript:void(0)). Request**header lines** ([line 3](javascript:void(0)) and [line 4](javascript:void(0))) are usually created automatically, so we don't have to worry about them. The **body**of the request on [line 5](javascript:void(0)) is what we will need to add to our POST.

response = requests.post("http://codecademy.com/learn-http/", data = body)

**Endpoints**

**Endpoints** are API-defined locations where particular data are stored. Just as you'll GET a pair of pants from PantsWorld or you'll GET a bag of peanuts from PeanutHut, you'll GET something different depending on the endpoint you use.

For instance, if you're using the API for a video hosting service, there might be endpoints for the most popular videos, the most recent videos, or videos belonging to a certain genre or category.

**Authentication & API Keys**

Many APIs require an **API key**. Just as a real-world key allows you to access something, an API key grants you access to a particular API. Moreover, an API key identifies you to the API, which helps the API provider keep track of how their service is used and prevent unauthorized or malicious activity.

Some APIs require authentication using a protocol called **OAuth**. We won't get into the details, but if you've ever been redirected to a page asking for permission to link an application with your account, you've probably used OAuth.

API keys are often long alphanumeric strings. We've made one up in the editor to the right! (It won't actually work on anything, but when you receive your own API keys in future projects, they'll look a lot like this.)

api\_key = "FtHwuH8w1RDjQpOr0y0gF3AWm8sRsRzncK3hHh9"

**HTTP Status Codes**

A successful request to the server results in a **response**, which is the message the server sends back to you, the client.

The response from the server will contain a three-digit status code. These codes can start with a 1, 2, 3, 4, or 5, and each set of codes means something different. (You can read the full list [here](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)). They work like this:

1xx: You won't see these a lot. The server is saying, "Got it! I'm working on your request."

2xx: These mean "okay!" The server sends these when it's successfully responding to your request.

3xx: These mean "I can do what you want, but I have to do something else first." You might see this if a website has changed addresses and you're using the old one; the server might have to reroute the request before it can get you the resource you asked for.

4xx: These mean you probably made a mistake. The most famous is "404," meaning "file not found": you asked for a resource or web page that doesn't exist.

5xx: These mean the server goofed up and can't successfully respond to your request.

**Anatomy of a Response**

The HTTP response structure mirrors that of the HTTP request. It contains:

1. A response line ([line 2](javascript:void(0))), which includes the three-digit HTTP status code;
2. A header line or lines ([line 3](javascript:void(0))), which include further information about the server and its response;
3. The body ([line 5](javascript:void(0)) and [line 6](javascript:void(0))), which contains the text message of the response (for example, "404" would have "file not found" in its body).

# HTTP/1.1 200 OK

# Content-Type: text/xml; charset=UTF-8

# <?xml version="1.0" encoding="utf-8"?>

# <string xmlns="http://www.codecademy.com/">Accepted</string>

**Parsing XML**

XML (which stands for eXtensible Markup Language) is very similar to HTML—it uses tags between angle brackets. The difference is that XML allows you to use tags that *you make up*, rather than tags that the [W3C](http://www.w3.org/) decided on. For instance, you could create an API that returns information about a pet:

<pet>

<name>Jeffrey</name>

<species>Giraffe</species>

</pet>

As long as you document the structure of your API's response, other people can use your API to get information about <pet>s.

from xml.dom import minidom

f = open('pets.txt', 'r')

pets = minidom.parseString(f.read())

f.close()

names = pets.getElementsByTagName('name')

for name in names:

print name.firstChild.nodeValue

**Parsing JSON**

JSON (which stands for JavaScript Object Notation) is an alternative to XML. It gets its name from the fact that its data format resembles JavaScript objects, and it is often more succinct than the equivalent XML. For instance, our same <pet> Jeffrey would look like this in JSON:

{

"pets": {

"name": "Jeffrey",

"species": "Giraffe"

}

}

Look, ma! No tags!

import json

from pprint import pprint

f = open('pets.txt', 'r')

pets = json.loads(f.read())

f.close()

pprint(pets) #pretty print

**XML or JSON?**

This leads us to wonder, though: how do we know whether an API will reply with XML or JSON?

The only way you'll know what type of data an API will send you is to read that API's documentation! Some will reply with one, and some will reply with the other. Documentation is a programmer's best friend, and it's always in your best interest to read it so you understand that what the API expects from you and what the API intends to send you when you make a request.

**find the total number of clicks on a link**

You can always see how many people have clicked on any bitly-powered link by taking the link and putting a plus sign (+) at the end of it. For example: <http://bitly.com/RYYpZT+>

**Find Referrers for a Link**

The 'referrer' is the site that contained the link that you clicked on to get to the current page. You can share bitly links on any site or social network, and then look at the referrers to figure out which sites are actually sending traffic to the link.