CS253 Unit 5

Web applications as services, using APIs

Contents

Introduction

Welcome to Unit 5. This unit will talk about how to make your web application talk to other computers.

Up until now, your programs have basically generated HTML that a browser will use to render a web page for the user. But your web application can also generate things like XML and JSON that other computers can interpret so that they, in turn, can build websites or services that interact with your data.

We will look at both XML and JSON and we'll see how to read them, how to interpret them and how to manipulate them. Then we will add a new feature to our ASCII Chan applications, that is built on Google Maps, which lets us see where our users are posting from.

Finally, your homework for this unit will be to make your blog output itself in JSON.

HTTP Clients

Until now, we've been working with this kind of standard picture of this little guy and his browser. Now, this used to be you, but now this is the user. You have upgraded to the other side of the picture where the servers are. So let's get these boxes. We've seen these 100 times. Okay, and let's add you over here. You are now the programmer. Congratulations, you are a web developer. We can talk about users and how much trouble they cause us.

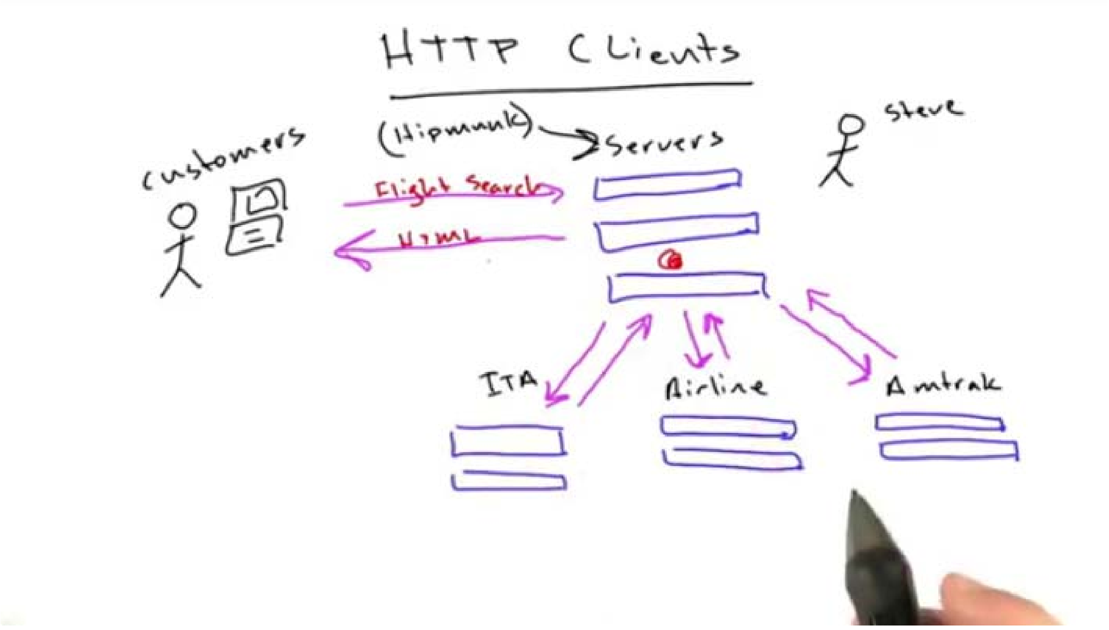
In the 'normal' web requests we have seen up until now, the user makes a request to the servers, and the server responds with the response. So far so good.

What we are going to talk about today is when your servers start making requests to other servers. Let's say that we have created a web application that does some data analysis on Twitter. Twitter has their own web servers, so we can have a web page that actually makes requests to Twitter. Our computers are now talking to their computers. This actually happens all the time. They're still communicating over HTTP, and Twitter will still respond as normal.

So, the user might make a request to us. We might then make a request to the Twitter servers, which respond with their normal response. Then we manipulate that data and return it to the user.

Hipmunk does a lot of this type of communication. Let's take a little look at how Hipmunk works.

When a user does a flight search on Hipmunk, they send requests to a bunch of data providers who have the actual flight data. Hipmunk takes the user's flight search and sends it to ITA, sends it to one or more airlines, and in some cases they'll even send it to Amtrak, if that's appropriate. Each of these organisations are companies who have their own services that Hipmunk works with. ITA will run the flight search, and send the results back. The Airline will also run the flight search on their own system and send their results back. Similarly, Amtrak does its thing and send their data back to Hipmunk. So, now Hipmunk has all this flight search data on their server. They then manipulate that data, collate it, make you nice results, and then we'll send back their HTML response to the user (or 'customer' in this case – since they're paying!).



So, this unit is about how to make our servers speak to other servers when there's no browser involved. We'll still use HTTP, but we are now communicating over other protocols. We saw some of this in Unit 1, but we'll be doing a lot more in this lesson, because there's so much you can do when you realize that you're not the only service on the internet.

urllib

First, let's introduce a Python library for actually making an HTTP requests so we can see how that works

Python has a library called urllib2. There's also a library called just urllib (referred to as "urlib1"), which does have a few handy functions of its own. However, we're going to use urllib2 for the most part.

urllib2 includes a function called urlopen. Let's say we want to download Google's homepage and store it in a variable p, for page. We can do this by entering the following code into the Python interpreter:

```
>>> import urllib2
>>>
>>> p = urllib2.urlopen("http://www.google.com")
>>>
>>> c = p.read()
```

p is a Python file object, which is basically an object with a read method. We called a read on p to get the contents of the page which we stored in c.

If we were to simply print c we would see a screen of text which is the code used to generate Google's front page. This is what we would expect. In fact, we have done the same thing before using telnet or curl. Now we can also do it using Python.

Now we have this variable c which contains the whole response, we can manipulate it in our programs. We are going to be doing a lot of this.

Let's take a look at what is held in the page, p. We can use the built-in dir() function to examine a Python object, thus:

```
>>> dir(p)
['__doc__',
'__init__',
'__iter__',
'__module__',
'__repr__',
'close',
'code',
'fileno',
'fp',
'getcode',
'geturl',
'headers',
'info',
'msg',
'next',
'read',
'readline',
'readlines',
'url']
>>>
```

These are the methods and attributes of our p object. We can immediately see that a couple of them that are probably going to be of interest to us. “headers” is one, and “geturl” is another. “getcode” is probably the status code.

This is often a good way to get to understand a library that you don’t know that well. Use dir() to examine the object.

Let's look at a couple of these. If we look at “url” we see:

```
>>> p.url
'http://www.google.co.uk/'
```

Which is the url that we requested. Looking a the “headers” gives us:

```
>>> p.headers
<httplib.HTTPMessage instance at 0x026D7CB0>
```

This is an HTTP message instance.

Now, I happen to know that this is a dictionary, and dictionaries have a function on them called items() which you can call on any dictionary to view the keys and values:

```
>>> p.headers.items()
[('x-xss-protection', '1; mode=block'),
('set-cookie',
'PREF=ID=20cb9ed43db0ec93:FF=0:TM=1337091686:LM=1337091686:S=Y6hHS34KbQHiPMGZ;
expires=Thu, 15-May-2014 14:21:26 GMT; path=/; domain=.google.co.uk,
NID=59=auTR3huH1WH0z9pfwue9E4Qk6MLwpphvjECKqMMdOtP-
RXmfz7C_UcLfndBJr72aMtJkDnqq5uOR5TdqWZhfWosIIWUXkBTCUKy0_uFggkNdiAr0HCKg_V
Aha--ez6kX; expires=Wed, 14-Nov-2012 14:21:26 GMT; path=/; domain=.google.co.uk; HttpOnly'),
('expires', '-1'),
('server', 'gws'),
('connection', 'close'),
('cache-control', 'private, max-age=0'),
('date', 'Tue, 15 May 2012 14:21:26 GMT'),
('p3p', 'CP="This is not a P3P policy! See
http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657 for more info."'),
('content-type', 'text/html; charset=ISO-8859-1'),
('x-frame-options', 'SAMEORIGIN')]
>>>
```

These are all of the headers we got back from Google. Since this is an actual dictionary, so if we want to know the content type, we enter:

```
>>> p.headers['content-type']
'text/html; charset=ISO-8859-1'
>>>
```

and we can see we're getting an ISO charset.

So, in future, especially for Windows users who may have had trouble using telnet, you can just use urllib and get the same answer.

Let’s try out what we just learned in a quiz.

Quiz: Using urllib

What server does www.example.com use?

Use Python and urllib2 and read the server header.

Redirection

I saved the page for www.example.com in a variable called p2. If we look at the url attribute on p2 we see:

```
>>> p2.url
'http://www.iana.org/domains/example/'
>>>
```

The url is actually iana.org. This is because example.com redirects to iana.org (as we saw back in unit 1). Now, urllib2 automatically follows redirects, and this can sometimes be confusing. In this case, it automatically followed the redirect to iana.org, whose server is Apache.

But if you do this by hand, as we did in unit 1, you find that the server is actually Big IP.

It is important for you to be aware when you use these libraries, that a lot of them, including urllib2 and the default one included in Google App Engine (called URL Fetch), follow redirects automatically. If you don't want to follow the redirects, there is almost always an option to tell the library not to automatically follow redirects.

XML

Let’s talk a little bit about what is actually sent over the wire between two computers.

We could have our servers make requests to other systems and receive the response as HTML. In fact, this is just what many servers do, but this is actually less than ideal.

We have seen HTML already, and by now you will know that it can be somewhat complex and not very regular. Fortunately, most browsers are very forgiving. Consider this HTML:

```
<form>
   <b>
      Hello!
</form>
```

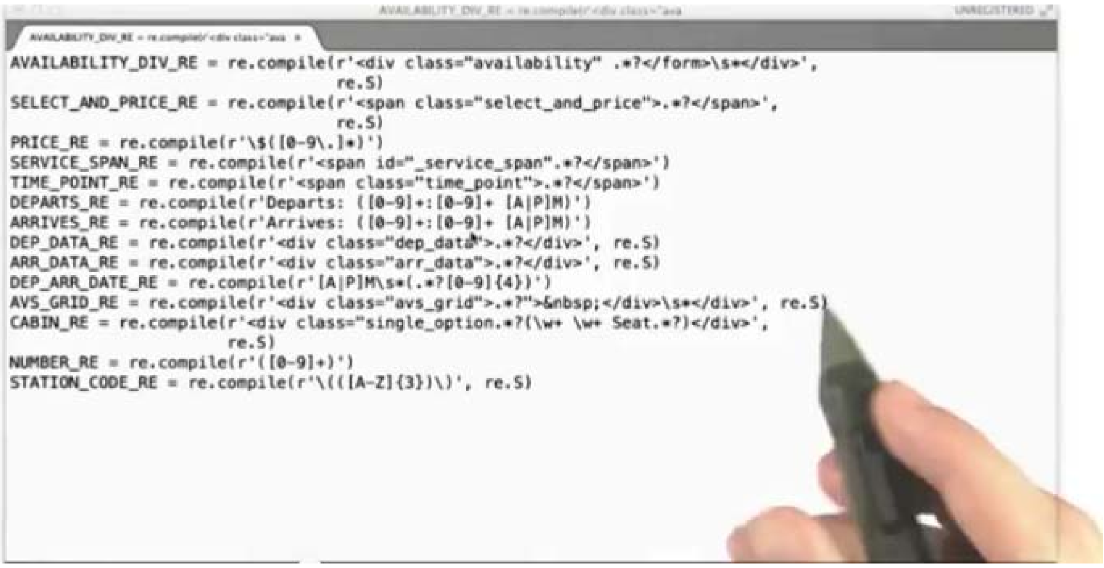
The closing </b> is missing. However, many browsers will still render this correctly.

A computer attempting to parse this code, will expect an opening <b> tag to have a closing </b> tag. If the closing tag is missing, the computer could get lost in this loop. You may be able to recover from this, like a browser does, depending on how complicated you want to make your parser. Then again, you may not.

So HTML is not really an ideal language for computer-to-computer communication.

Some servers do use HTML. When Hipmunk sends a request to Amtrak they actually get HTML from Amtrak. They then have a fairly involved process to parse this.

We used some regular expressions to validate usernames and emails in Homework 2. Hipmunk use similar regular expressions when they parse HTML from Amtrak:



```
AVAILABILITY_DIV_RE = re.compile(r'<div class="availability" .*?</form>\s*</div>',
                                 re.S)
SELECT_AND_PRICE_RE = re.compile(r'<span class="select_and_price">.*?</span>',
                                 re.S)
PRICE_RE = re.compile(r'\$([0-9\.]*)')
SERVICE_SPAN_RE = re.compile(r'<span id="_service_span".*?</span>')
TIME_POINT_RE = re.compile(r'<span class="time_point">.*?</span>')
DEPARTS_RE = re.compile(r'Departs: ([0-9]+:[0-9]+ [A|P]M)')
ARRIVES_RE = re.compile(r'Arrives: ([0-9]+:[0-9]+ [A|P]M)')
DEP_DATA_RE = re.compile(r'<div class="dep_data">.*?</div>', re.S)
ARR_DATA_RE = re.compile(r'<div class="arr_data">.*?</div>', re.S)
DEP_ARR_DATE_RE = re.compile(r'[A|P]M\s*(.*?[0-9]{4})')
AVS_GRID_RE = re.compile(r'<div class="avs_grid">.*?">&nbsp;</div>\s*</div>', re.S)
CABIN_RE = re.compile(r'<div class="single_option.*?(\w+ \w+ Seat.*?)</div>',
                      re.S)
NUMBER_RE = re.compile(r'([0-9]+)')
STATION_CODE_RE = re.compile(r'\(([A-Z]{3})\)', re.S)
```

```
AVAILABILITY_DIV_RE = re.compile(r'<div class="availability" .*?</form>s*</div>',
                                 re.S)
SELECT_AND_PRICE_RE = re.compile(r'<span class="select_and_price">.*?</span>',
                                 re.s)
PRICE_RE = re.compile(r'$([0-9.]*)')
SERVICE_SPAN_RE = re.compile(r'<span id="_service_span".*?</span>')
TIME_POINT_RE = re.compile(r'<span class="time_point">.*?</span>')
DEPARTS_RE = re.compile(r'Departs: ([0-9]+:[0-9]+ [A|P]M)')
ARRIVES_RE = re.compile(r'Arrives: ([0-9]+:[0-9]+ [A|P]M)')
DEP_DATA_RE = re.compile(r'<div class="dep_data">.*?</div>', re.S)
ARR_DATA_RE = re.compile(r'<div class="arr_data">.*?</div>', re.S)
DEP_ARR_DATE_RE = re.compile(r'[A|P]M\s*(.*?[0-9]{4})')
AVS_GRID_RE = re.compile(r'<div class="avs_grid">.*?">&nbsp;</div>\s*</div>' re.S)
CABIN_RE = re.compile(r'<div class="single_option.*?(\w+ \w+ Seat.*?)</div>',
                      re.S)
NUMBER_RE = re.compile(r'([0-9]+')
STATION_CODE_RE = re.compile(r'(([A-Z]{3})\)', re.S)
```

At first glace, this appears to be just a wall of text. As you can imagine, this is extremely error-prone, and is not the ideal way of doing things..

In an ideal world, we would use an API (application programming interface) that uses a language more appropriate for the task. Such a language might be XML.

XML was actually invented in the late '90s specifically to enable a regular way of expressing data between computer systems. This is what some XML might look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<results>
  <itinerary>
    <leg>
      <origin>CUS</origin>
      <dest>WAS</dest>
    </leg>
  </itinerary>
</results>
```

If you think this looks an awful lot like HTML, you are correct!

The first line is basically the document type. We have the same thing in HTML. We've been using HTML5, so our HTML doctype also looks something like this. It's just the first line that says what format the rest of the document is.

Now, the reason both HTML and XML look so similar is that they share a common ancestor in SGML (Standard Generalised Markup Language), which was invented in the 1980s.

The main difference between XML and HTML is that in XML every tag has to have a closing tag. The tag format is the same with opening tags, <results>, and closing tags, </results>, but **there are no void tags in XML**.

In HTML we have the <br> tag to put in a line break. This doesn't need a closing tag because HTML doesn't require all tags to close. We have the notion of a void tag, and the line break is an example of one of those.

XML has nothing like that . Now, if you want a tag that has no content in XML, you could include a closing slash before the greater-than symbol, something like this:

```
<br/>
```

In fact, there is a doctype for HMTL called "XHTML":

```
<!DOCTYPE xhtml>
```

This basically says that the HTML document is actually also going to be valid XML. In this case, instead of void tags with no closing slash, the closing slash is included before the greater-than.

In essence, XML is very similar to HTML, but it is more rigorous.

Quiz: XML and HTML

Which of these are true statements?

- All HTML is XML?
- All XML is HTML?
- HTML can be expressed in XML?
- XML and HTML share a common lineage.

Select all that are correct.

Parsing XML

So, how do we parse XML?

Python includes a built-in parser. Python has a library called "minidom". You can use it by running something like:

```
from xml.dom import minidom
```

The term “DOM” is something you’ll see quite often when you work with XML. This stands for "Document Object Model", and essentially refers to the internal computer representation of an XML document. In Python, the XML document is an object that has a list of children. Each of these children is some sort of tag object, and these tag objects may have attributes like name, contents and so on.

Actually parsing XML is really complicated. XML can be many, many gigabytes in size. Parsing all of that text is definitely nontrivial! However, if you're only parsing a small amount of text, you can use the minidom library, which is a smaller, ‘lightweight’ version of the DOM parser that is simple and fast but which will break if you throw really large amounts text at it. For our purposes in this unit, minidom will work just fine.

Let’s take a look at minidom in action. Now, minidom has a function called parseString(), which will parse a string of XML. We can enter the code into the Python interpreter as shown below:

```
>>> from xml.dom import minidom
>>>
>>> minidom.parseString
<function parseString at 0x02853DB0>
>>> x = minidom.parseString("<mytag>contents!<chilrdren><item>1</item><item>2</item></children>
         </mytag>")
```

Looking at the XML string, we have an opening tag, <mytag>, followed by some text, and then an opening <children> tag and some items within <item></item> tags, and then the closing </children> and </mytag> tags.

There are a couple of things to note here. Firstly, these are just made up tag names. In HTML, there are specific tags that you have to use. With XML you can use whatever tags you want to use, as long as the people writing and reading the XML agree on the meaning of the names.

Secondly, we have included a deliberate typo in the opening <chilrdren> tag. If we run the code as shown, we get an error:

```
        ExpatError: mismatched tag: line 1, column 57
```

A mismatched tag. Well, that does make sense!

If we run the code without the typo we will get a minidom document instance:

```
>>> x
<xml.dom.minidom.Document instance at 0x027B3A08>
>>>
```

We can use the dir() function to see what is in x:

```
>>> dir(x)
['ATTRIBUTE_NODE',
'CDATA_SECTION_NODE',
'COMMENT_NODE',
'DOCUMENT_FRAGMENT_NODE',
'DOCUMENT_NODE',
'DOCUMENT_TYPE_NODE',
'ELEMENT_NODE',
'ENTITY_NODE',
'ENTITY_REFERENCE_NODE',
'NOTATION_NODE',
'PROCESSING_INSTRUCTION_NODE',
'TEXT_NODE',
'__doc__',
'__init__',
'__module__',
'__nonzero__',
'_call_user_data_handler',
'_child_node_types',
'_create_entity',
'_create_notation',
'_elem_info',
'_get_actualEncoding',
'_get_async',
'_get_childNodes',
'_get_doctype',
'_get_documentElement',
'_get_documentURI',
'_get_elem_info',
'_get_encoding',
'_get_errorHandler',
'_get_firstChild',
'_get_lastChild',
'_get_localName',
'_get_standalone',
'_get_strictErrorChecking',
'_get_version',
'_id_cache',
'_id_search_stack',
'_magic_id_count',
'_set_async',
'abort',
'actualEncoding',
'appendChild',
'async',
'attributes',
'childNodes',
'cloneNode',
'createAttribute',
'createAttributeNS',
'createCDATASection',
'createComment',
'createDocumentFragment',
'createElement',
'createElementNS',
'createProcessingInstruction',
'createTextNode',
'doctype',
'documentElement',
'documentURI',
'encoding',
'errorHandler',
'firstChild',
'getElementById',
'getElementsByTagName',
'getElementsByTagNameNS',
'getInterface',
'getUserData',
'hasChildNodes',
'implementation',
'importNode',
'insertBefore',
'isSameNode',
'isSupported',
'lastChild',
'load',
'loadXML',
'localName',
'namespaceURI',
'nextSibling',
'nodeName',
'nodeType',
'nodeValue',
'normalize',
'ownerDocument',
'parentNode',
'prefix',
'previousSibling',
'removeChild',
'renameNode',
'replaceChild',
'saveXML',
'setUserData',
'standalone',
'strictErrorChecking',
'toprettyxml',
'toxml',
'unlink',
'version',
'writexml']
>>>
```

There is a lot of interesting stuff here .

There are functions for manipulating the document, like appendChild() and createTextNode(). There are lookup functions like getElementById(), and getElementByTagName(). We will be using these a lot more later.

An intersting function is Toprettyxml(). If we take our document object, x, and call "toprettyxml" on it we get:

```
>>> x.toprettyxml()
u'<?xml
version="1.0" ?>\n<mytag>\n\tcontents!\n\t<children>\n\t\t<item>\n\t\t\t1\n\t\t</item>\n\t\t<item>\n\t\t
\t2\n\t\t</item>\n\t</children>\n</mytag>\n'
>>>
```

This really doesn't look very pretty at all, but this is actually the Python string with the new lines in it, and if we print it we get:

```
>>> print x.toprettyxml()
<?xml version="1.0" ?>
<mytag>
        contents!
        <children>
                <item>
                        1
                </item>
                <item>
                        2
                </item>
        </children>
</mytag>
>>>
```

Which is much more readable. Here is the xml that we entered, and we can see the structure of the document nicely indented for us. This is really a handy little function. When you download XML from somewhere it can be much easier to see the structure of it if you use prettyxml().

Another function I'd like to show you here is getElementByTagName(). If we run this on our document object, x, and give it “mytag” it will return one DOM element:

```
>>> x.getElementsByTagName("mytag")
[<DOM Element: mytag at 0x27b3b98>]
>>>
```

If we run it for “item”, we actually get two DOM elements:

```
>>> x.getElementsByTagName("item")
[<DOM Element: item at 0x269a558>, <DOM Element: item at 0x269a4e0>]
>>>
```

If we look at the children of the first item, we can access the node value attribute and see that it is 1:

```
>>> x.getElementsByTagName("item")[0].childNodes
[<DOM Text node "u'1'">]
>>>
```

In effect, what we have just done was to say “get me all of the elements that are called item”. We got two results. On the first of these we said “get me its first child”, which has the value “1”. Now, this isn't strictly a node, but in minidom, it is represented as a text node.

Different libraries may handle contents differently, but in minidom this is how we get it. We can actually get the value of that text node:

```
>>> x.getElementsByTagName("item")[0].childNodes [0].nodeValue
u'1'
>>>
```

The value is the number 1. The “u” means a unicode string since minidom assumes that we entered a unicode string.

RSS

You've probably at least heard of RSS. It's a way to read a website that has daily content like a blog or a news site. You may even have a specialised reader for reading the content.

RSS originally stood for "**RDF Site Summary**". RDF stands for "**Resource Description Framework**". RDF is an XML format which can be used to describe just about anything. It's basically a way to represent knowledge in XML.

More commonly these days, RSS is dubbed "**Really Simple Syndication**". This is more the context in which we're going to be using it. RDF was originally conceived as a tool to organize the world's information problems, and RSS uses RDF, but in essence, it's just a list of content in XML.

Let’s look at some examples of RSS in the wild.

The New York Times homepage has a link for RSS at the bottom of this page. This will take us to a page of links for various NYT news sites. If we then click on the link for **NYTimes.com Home Page (Global Edition)** we see that the URL includes GlobalHome.xml:

http://www.nytimes.com/services/xml/rss/nyt/GlobalHome.xml

We have received an XML document.

Most browsers do display XML in a nice way. Although this particular document is an XML document, it is actually an RSS feed. What this means is that there is a particular ‘namespace’ (something like a tag-space) for the items in the list that make up the feed.

Just like an HTML document opens with an HTML tag and has a header and a body with specific tags, an RSS document also has very specific tags. In XML, you can use the header area to describe the namespace that you are going to use. This is the header from GlobalHome.xml:

```
<?xml version="1.0"?>
<rss xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:media="http://search.yahoo.com/mrss/"
xmlns:atom="http://www.w3.org/2005/Atom"
xmlns:nyt="http://www.nytimes.com/namespaces/rss/2.0" version="2.0">
```

The page is using the atom name space and the RSS 2.0 name space. This tells the parser that descriptions of the tags can be downloaded from these URLs, and so what tags to expect.

If you look at the XML file for this page, you’ll see that there is some initial header information followed by a list of items which are the stories in the feed. Each item has a title and a link, a brief description and so forth. Exactly the things that are needed by an RSS reader or similar program to download the contents of The New York Times without having to parse all the HTML.

Quiz: Parsing RSS

How many <item> elements are in the page listing at the New York Times Global Edition RRS feed:

http://www.nytimes.com/services/xml/rss/nyt/GlobalHome.xml

Use urllib and minidom in Python to download this page. Remember the function getElementsByTagName( ) will be particularly useful to you on the minidom object.

JSON

The next thing we need to introduce is JSON. JSON is another computer and human readable way to exchange data in a consistent format. JSON stands for “JavaScript Object Notation”, and JSON is, in fact, valid JavaScript code.

Sticking with our earlier travel search example, JSON may look something like this:

```
{"itineraries": [{"from":"SCO",
                  "to": "IAD"},
                 {"from": "IAD",
                  "to": "SEA"}]
        }
```

This dictionary structure actually looks a lot like Python code. This is because Python and JavaScript have very similar syntax for dictionaries and lists.

This dictionary might have a key called "itineraries" whose value is a list of other dictionaries. Here we have a dictionary for each routing or leg of the journey, denoted by the curly-braces, which has a key for "from" and a key for "to", each of which has a value.

JSON is really useful if you want to express object of these types. Anything you can express in XML, you can also express in JSON. JSON is a little less verbose, because you don't need all the opening and closing tags.

The main data structures in JSON are dictionaries (or “mappings”), which match a key to a value, or multiple keys to multiple values, and lists.

Dictionaries are just a list of key-value pairs enclosed by curly-braces. Just as in Python. Lists use square-brackets, just like Python does, and separate the values in the list with commas.

Dictionaries	**Lists**
{“key”: value}	[1, 2, “three”, [True, 5.6]

The data types available in JSON are: integer, string, boolean, and float. We can, of course, also have "null," as in the empty list [] for example.

Let's look at how to parse JSON in Python. JSON is now included in Python version 2.6 and later (App Engine currently uses version 2.7).

We begin by importing JSON and then making a JSON string in Python, which we will call j:

```
>>> import json
>>> j = '{"one": 1, "numbers": [1,2,3.5]}'
```

Our JSON string is a dictionary with two keys, "one" and "number". The value of "one" is 1, and the value of "numbers" is the list [1, 2 , 3.5].

JSON includes the function loads(), which stands for "load string" which parses the string:

```
>>> json.loads(j)
{u'numbers': [1, 2, 3.5], u'one': 1}
```

This has returned a Python dictionary with the same keys, "numbers" and "one" and the same values, 1 and [1, 2, 3.5]. The order doesn't matter in a Python dictionary.

If we store this as a variable d, we can then manipulate it. We can look at just d['numbers'] to see our list. :

```
>>> d = json.loads(j)
>>>
>>> d['numbers']
[1, 2, 3.5]
```

We can look at d['one'], and we'll see our number 1:

```
>>> d['one']
1
```

In fact, because JSON looks just like Python, we could even eval(j):

```
>>> eval(j)
{'numbers': [1, 2, 3.5], 'one': 1}
```

What eval() does is to treats the argument as Python code as if it had just been typed at the prompt. Now, although that's a pretty neat thing you can do, **you should never, ever do it**. This is because, in addition to having JSON in here, somebody could actually have malicious code that might do something to your computer. **Never use eval() for parsing JSON.**

Quiz: Parsing JSON

reddit_front is a JSON string of reddit's front page. Inside it is a list of links, each of which has an "ups" attribute. From this dataset, what is the total number of ups of all the links?

Implement the function total_ups(), and make it return the total number of ups. This is going to require some experimental searching through the reddit_front JSON, which is a fairly typical problem when dealing with APIs, RSS, or JSON.

You'll need to load the json using the json.loads method, after which you should be able to search through the json similarly to a dictionary or list.

Note that you will need to access parts of the JSON as a dictionary, and others as a list.

APIs

We can access any page on Reddit in JSON or in XML just by changing the extension. The required extensions are simply .json or .xml respectively.

A lot of web-pages now have this feature allowing you to get their content in different formats. A good example of this is Twitter.

If we do a search for Udacity on Twitter, we can see all the tweets about Udacity. If we want this listing in JSON, we can just go to:

search.twitter.com/search.JSONq=Udacity

The result is the JSON listing of the same search result that we were just looking at on Twitter.

If you're writing web software and you want to manipulate another website or just get data from another website, it is usually not very difficult to find their APIs. We found this API by just googling "Twitter API", and the documentation for this came up. There's a whole page about Twitter's search API:

https://dev.twitter.com/docs/api/1/get/search

If you are building websites that have data, such as a blog, for example, designing your site so that it supports RSS, an XML representation of its content and a JSON representation of its content, allows other people to build software on top of your website that can do cool things.

Many large websites, including Wikipedia, Twitter, and Reddit support these types of functionality. Your homework in this class is actually going to be to add that functionality to the blog that you've been working on.

JSON Escaping

We saw in unit 2 how we had to escape any HTML content so that it rendered correctly in the browser. There is a similar issue with JSON.

Imagine that we have a s little JSON blog with a dictionary that takes a key "story" and maps it to the string "once upon a time:

```
{"story": "once upon a time"}
```

This is valid JSON. In this case, the key is surrounded by double quotes, and the value is also surrounded by double quotes.

What would happen if the value also happened to include double quotes? Let's say for argument's sake that the dictionary was actually:

```
{"story": "once u"on a time"}
```

This would cause a problem, because this is now invalid JSON. As far as the JSON parser is concerned, the quote we inserted now ends the string, and the string is followed by some garbage characters.

We can get around this problem by escaping the extra double quote. We can escape the quote in Python by just adding a slash in front for it:

```
{"story": "once u\"on a time"}
```

Now, this escapes the quote in Python, but what this turns into for the JSON parser is just the string that we had before., i.e.

```
{"story": "once u"on a time"}
```

In fact, we need to escape both the quote and the slash by adding a second slash in order for this to work:

```
{"story": "once u\\"on a time"}
```

Basically, the second slash is the Python escape for the first slash. This tells Python that we're inserting a slash into the string, and yes, we actually meant to do that. Now, the JSON interpreter will see that slash and understand that we did mean to include the quote.

There is another way to achieve the same result in Python which can be a little simpler. The double slashes can be a little confusing, so what we do is just put an "r" in front of the string. This indicates that it is a raw string and tells Python to ignore any escaping we're doing here:

```
json.loads(r'{"story": "once u\"on a time"}')
```

The JSON module can now interpret the slash and escape the additional quote.

Escaping shouldn’t be a major issue when we read JSON, because we assume that the JSON we're going to be reading is valid. If it isn’t valid, our JSON module will tell us by throwing an exception when we try to read it.

The function for writing JSON in Python is called dumps(). This stands for “dump-string”. You can pass a Python object into dumps() and it will convert that object into JSON for you. Let’s see that in the Python interpreter. We will pass a list, [1, 2, 3] into dumps():

```
>>> json.dumps([1, 2, 3])
'[1, 2, 3]'
>>>
```

We didn’t put quotes around the list because this is the actual list object we want to convert into JSON. The output is the string, '[1, 2, 3]'. What happens if we try this with a more complicated object? Let’s try with the dictionary object we saw earlier, {"one": 1, "two": 2}:

```
>>> json.dumps({"one": 1, "two": 2})
'{"two": 2, "one": 1}'
>>>
```

Now the order has changed, but then the order isn’t defined in dictionaries, and we now have our JSON. Let’s change the dictionary so that the second value needs to be escaped, {"one": 1, "two": 'the man said, "cool!"'}, and pass this into dumps():

```
>>> json.dumps({"one": 1, "two": 'the man said, "cool!"'})
'{"two": "the man said, \\"cool!\\"", "one": 1}'
>>>
```

The JSON library has escaped the extra double-quotes for us, and has produced a Python sting that is a representation of valid JSON.

A word of caution: the string produced in the last example is not, in itself, valid JSON. It is valid Python *representing* valid JSON. That's why we had those double slashes. You need to be careful when you are copying and pasting JSON in and out of Python and make certain that you get the escaping right.

If we print the last expression in the Python shell we get:

```
>>> print json.dumps({"one": 1, "two": 'the man said, "cool!"'})
{"two": "the man said, \"cool!\"", "one": 1}
>>>
```

The string: {"two": "the man said, \"cool!\"", "one": 1}, with the single slashes is the actual valid JSON.

One other thing to remember is that the main distinction between a Python object and a JSON representation of a very similar object is that **JSON has to use double quotes to delineate a string**. You can't use single quotes like can with Python.

Now, as we've seen, JSON maps very nicely to Python data structures, provided that you're using integers, floats and strings. If you want to map some of the more complex Python structures to JSON (say you need to map an object or a date time), the simple JSON dumps() function isn't going to work. You will have to convert these by hand to simple data structures made up of dictionaries and lists of integers and strings so that they can be ouput properly.

Quiz: Escaping JSON in Python

What is the valid JSON for this Python datastructure:

```
{"blah":["one", 2, 'th"r"ee']}
```

Note that JSON must use double quotes to enclose strings, it cannot use single quotes.

Being a Good Citizen

Let's take a few moments to talk about how to be a good citizen on the internet.

When you write programs to access or manipulate data on other people's websites there are a couple of things you can do that make life easier for everyone. Firstly, always use a good user-agent. You'll remember from Unit 1 that user agents are the header that describes what browser you're using, or what program you're using to access a site.

If you use urlLib2, you can specify headers in your request. You should set a user agent header that says who you are, what you're name is, and perhaps a link to your website. This is not only good Internet manners, it is in your own interests.

If you're going to be accessing a site relatively heavily or consistently, perhaps checking for regular updates or something of that sort, when they see you hitting them with lots of requests they can check your user agent header to find out who you are, and perhaps why you are accessing their site. If your requests are hurting their server response times, they can contact you and ask that you stop or change your access pattern rather than simply blocking you.

The second really important thing you should do is to rate-limit yourself. Let's say that you wanted to download all the search results for the word Udacity on Twitter. You can request them from Twitter and their API will return 15 at a time. You could set up a loop and send requests to Twitter as fast as their servers can handle them until you've downloaded all the search results. The problem is that this would actually hurt the performance of Twitter's servers. The solution is to slow the speed at which you send requests.

In Python, you can import the time library and use the sleep() function to introduce a delay into your code:

```
import time
while more:
   get_more()
   time.sleep(1)
```

This will cause the Python interpreter to sleep for one second, and so introduces a one second delay. Now you'd only be hitting their servers once per second, which is much more sustainable. If you abuse their service or send too many requests, they'll probably rate-limit you. Twitter certainly will!

SOAP

There are a number of other common protocols and formats for communicating across the Internet. Lots of people use these protocols, and if you need to access servers that use these protocols, then you will have to use them too. We will list some of these here.

The first of these is SOAP, or Simple Object Access Protocol This is another protocol for communicating between two machines. It is based on XML and is very, very complicated. It was invented by Microsoft to make online communication as complicated as possible. A lot of the data sources used by Hipmunk communicate via SOAP.

Next are Protocol Buffers, which are Google's data interchange format. It's similar in concept to JSON, and is a way of encoding different types of data for sending it over the wire.

Apache Thrift is used by Facebook. This is a “software framework, for scalable cross-language services development”.

Protocol Buffers and Thrift might be compared more to JSON. SOAP would compare more to HTTP plus JSON as a complete package of the protocol and the data type. Implementations of JSON, SOAP, Thrift and Protocol buffers can be found in almost any language.

There are also a whole range of plaintext, custom formats. You can always just build your own plaintext protocol, and data format, but this isn’t generally recommended. It's really not that hard to use JSON instead. Then, when somebody who needs to use the service comes along, they don't have to figure out how to write all the custom code needed to parse your custom stuff.

As a general rule, use something that already exists. It'll save everybody a lot of time.

Quiz: Good Habits

Which of these are good habits to get into?

- Sending proper user agents?
- Writing custom protocols?
- Using SOAP?
- Rate-limiting yourself?
- Using common protocols and data formats?

ASCII Chan 2

Let’s get on and do some actual programming.

We're going to add a new feature to ASCII Chan. What we are going to do is to add a map to the front page that shows where the most recent submissions came from. This way, when you come to ASCII Chan you will see what a global community it is.

To achieve this, we have to do a number of things things.

First, we are going to need to figure out where the user submitting the art is from. What we need is a service that converts the request IP address into map coordinates. We are going to use a handy little service called hostip.info. Basically, for any IP address, they will tell you where it's located. They also have an API that has very simple documentation. The documentation is located here.

In essence, if we go to api.hostip.info and include the IP address in a get parameter:

    http://api.hostip.info/?ip=12.215.42.19

they'll return some XML with location data:

```
<HostipLookupResultSet version="1.0.1" xsi:noNamespaceSchemaLocation
                ="http://www.hostip.info/api/hostip-1.0.1.xsd">
  <gml:description>This is the Hostip Lookup Service</gml:description>
  <gml:name>hostip</gml:name>
  <gml:boundedBy>
        <gml:Null>inapplicable</gml:Null>
  </gml:boundedBy>
  <gml:featureMember>
        <Hostip>
                <ip>12.215.42.19</ip>
                <gml:name>Aurora, TX</gml:name>
                <countryName>UNITED STATES</countryName>
                <countryAbbrev>US</countryAbbrev>
                <!-- Co-ordinates are available as lng,lat -->
                <ipLocation>
                        <gml:pointProperty>
                                <gml:Point srsName="http://www.opengis.net/gml/srs/epsg.xml#4326">
                                        <gml:coordinates>-97.5159,33.0582</gml:coordinates>
                                </gml:Point>
                        </gml:pointProperty>
                </ipLocation>
        </Hostip>
  </gml:featureMember>
</HostipLookupResultSet>
```

This includes the country and some coordinate information (longitude and latitude).

The next thing we need to do is to draw a map. We'll use Google Maps for this.

Google Maps has a really useful service called "static maps" which allows us to embed a Google map on our page. The Google Static Map service will create our map based on URL parameters that we can send through a standard HTTP request, and will return the map as an image that we can display on our front page.

The documentation for Static Maps is on the Google Developers site.

Geolocation

Let's begin with the first task, which is to implement a function that looks up the IP address. This is our current ASCI Chan code:

```
import os
import re
import sys

from string import letters

import webapp2
import jinja2
from google.appengine.ext import db

template_dir = os.path.join(os.path.dirname(__file__), 'templates')
jinja_env = jinja2.Environment(loader = jinja2.FileSystemLoader(template_dir), autoescape=True)

art_key = db.Key.from_path('ASCIIChan', 'arts')

def console(s):
        sys.stderr.write('%s\n' % s)

class Handler(webapp2.RequestHandler):
        def write(self, *a, **kw):
                self.response.out.write(*a, **kw)

        def render_str(self, template, **params):
                t = jinja_env.get_template(template)
                return t.render(params)

        def render(self, template, **kw):
                self.write(self.render_str(template, **kw))

class Art(db.Model):
        title = db.StringProperty(required = True)
        art = db.TextProperty(required = True)
        created = db.DateTimeProperty(auto_now_add = True)
```

```
class MainPage(Handler):
	def render_front(self, title="", art="", error=""):
		arts = db.GqlQuery("SELECT * "
							"FROM Art "
							"WHERE ANCESTOR IS :1 "
							"ORDER BY created DESC "
							"LIMIT 10",
							art_key)

		self.render("front.html", title=title, art=art, error = error, arts = arts)

	def get(self):
		self.render_front()

	def post(self):
		title = self.request.get("title")
		art = self.request.get("art")
		if title and art:
			p = Art(parent=art_key, title = title, art = art)
			p.put()

			self.redirect("/")
		else:
			error = "we need both a title and some artwork!"
			self.render_front(error = error, title = title, art =art)

app = webapp2.WSGIApplication([('/', MainPage)],
                              debug=True)
```

Our get() function in the MainPage() handler just calls the function we called render_front(). This runs a basic query to get the 10 most recent pieces of art from the Google database object Art.

The post() function gets the title and the artwork from the request. If both of them are present, it creates a new art object and stores it in a database, and then reloads the page by doing a redirect. If the title and/or the artwork are missing, it displays an error message. We want to modify this to look up the user’s coordinates from their IP address and, if they are available, add them to the art object:

```
	def post(self):
		title = self.request.get("title")
		art = self.request.get("art")

		if title and art:
			p = Art(parent=art_key, title = title, art = art)
			#lookup the user's coordinates from their IP
			#if we have coordinates, add them to the art
			a.put()

			self.redirect("/")
		else:
			error = "we need both a title and some artwork!"
			self.render_front(error = error, title = title, art =art)
```

We know that we are going to request a URL and that we will be receiving some XML that we will need to parse, so we need to add the urllib2 and mindom libraries:

```
import os
import re
import sys
import urllib2
from xml.dom import minidom

from string import letters

import webapp2
import jinja2

from google.appengine.ext import db
```

We are going to need a function to look up the users coordinates from their IP address. We'll call this function get_coords(). This will take an IP address as its argument and make a request to hostip.info. We've taken the URL from the hostip.info. API documentation and removed the sample IP address:

```
IP_URL = "http://api.hostip.info/?ip="
def get_coords(ip):
        url = IP_URL + ip
```

Now, you already know how to make a basic URL request. We'll show you a few things you can do in the real world to make the requests a little less prone to errors.

We are going to store the content of the URL in a variable called "content". We'll be using urllib2.urlopen():

```
content = urllib2.urlopen(url).read()
```

If the URL is invalid or the website is down, this is going to generate an exception. In this case we happen to know what that exception is, so we can write:

```
def get_coords(ip):
        url = IP_URL + ip
        content = None
        try:
                content = urllib2.urlopen(url).read()
        except urllib2.URLError:
                return
```

If there is a URL error, there are no are no coordinates, and we'll just return.

Normally we would probably log these errors so that when we are maintaining the site we can see that the geocoding is broken for some reason.

Quiz: Geolocation

Implement the get_coords(xml) function that takes in an xml string and returns a tuple of (lat, lon) if there are coordinates in the xml.

Remember that you should use minidom to do this. Also, notice that the coordinates in the xml string are in the format: (lon,lat), so you will have to switch them around.

We have provided an example chunk of XML that comes from that website that should work.

Debugging

Here is our get_coords() function:

```
def get_coords(ip):
        url = IP_URL + ip
        content = None
        try:
                content = urllib2.urlopen(url).read()
        except URLError:
                return

        if content:
                d = minidom.parseString(content)
                coords = d.getElementsByTagName("gml:coordinates")
                if coords and coords[0].childNodes[0].nodeValue:
                        lon, lat = coords[0].childNodes[0].nodeValue.split(',')
                        return db.GeoPt(lat, lon)
```

We have slightly modified the code from the quiz solution. Google App Engine includes a data type for storing a latitude and longitude. This is GeoPt. So what we actually want to return is db.GeoPt(lat, long).

Let's run a quick test to check that this function is working. We are going to add a line to our get() function:

```
        def get(self):
                self.write(repr(get_coords(self.request.remote_addr)))
                self.render_front()
```

When you print a Python object, it has angle brackets around it. When you print it in HTML, the browser will interpret the object as a tag, and it won't actually print what you're trying to print. Using repr() is a handy trick when you need to print Python objects in HTML. You'll get some extra quotes and the object will print properly.

The remote_addr attribute of the request object that is the parameter for the get_coords() function is the requesting IP address. Almost every web framer will give you access to the requesting IP.

If we reload the web page we see the response in the browser:

```
None
```

Well, that's certainly better than an exception! Let's see if we can discover why we didn't get a location. Let's add the following line to our get() function:

```
self.write(self.request.remote_addr)
```

Now we see:

```
127.0.0.1None
```

The IP address 127.0.0.1 means that we're running the app locally. For any of you who don't know, every machine has a local IP address of 127.0.0.1. This is known as the **loop-back address**, and is how a machine refers to itself. Local host generally refers to that IP address. It's hardly surprising that a service on the internet doesn't know what our local IP is!

For the purpose of testing the get_coords() function, we can manually set the IP to an address we know is real: 4.2.2.2. This is a big-name server than helps resolve TNS names into IPs. The get_coords() function is now:

```
def get_coords(ip):
    ip = "4.2.2.2"
    url = IP_URL + ip
    content = None
    try:
        content = urllib2.urlopen(url).read()
    except URLError:
        return

    if content:
        d = minidom.parseString(content)
        coords = d.getElementsByTagName("gml:coordinates")
        if coords and coords[0].childNodes[0].nodeValue:
            lon, lat = coords[0].childNodes[0].nodeValue.split(',')
            return db.GeoPt(lat, lon)
```

Let's refresh our browser, and see if we get anything useful:

```
datastore_types.GeoPt(39.994, -105.062)
```

So we're getting the coordinates of the IP address where that machine is located. Our IP function is working and, as a bonus, we have also tested the error case (which is something you should always do).

Back to our to-do list.

Updating the Database

Now we are ready to modify our post function. The first thing we heed to add is a line to lookup the user's coordinates from their IP address:

```
coords = get_coords(self.request.remote_addr)
```

Next, if there are coordinates, we need to add them to the art. At the moment, our Art object doesn't take coordinates, so we'll have to add an extra property to our Art database. Since we're returning a Geo Point, we can store it in a Geo Point Property:

```
class Art(db.Model):
	title = db.StringProperty(required = True)
	art = db.TextProperty(required = True)
	created = db.DateTimeProperty(auto_now_add = True)
	coords = db.GeoPtProperty( )
```

Obviously, we'd like to say (required = True) for coords, but we already have some art in our database that doesn't have coordinates. There are a couple of options available to us.

We could just delete all that art and start over. However, since ASCII Chan is such a hot site on the internet, and since everybody is using it, we don't want it to just break.

Alternatively we could just make this parameter not required. We'll just have it for future art. This is what we will do in this case.

Actually. this is something that comes up all the time when you're developing web applications. Because you will often be adding features, or tweaking the data model for your sites, you'll need to make decisions like this on a regular basis. It's kind of backwards compatibility.

Now we're ready to add the coordinates, if they exist, to the art. This is easy enough:

```
def post(self):
        title = self.request.get("title")
        art = self.request.get("art")

        if title and art:
                p = Art(parent=art_key, title = title, art = art)
                #lookup the user's coordinates from their IP
                coords = get_coords(self.request.remote_addr)
                #if we have coordinates, add them to the art
                if coords:
                        p.coords = coords

                a.put()

                self.redirect("/")
        else:
                error = "we need both a title and some artwork!"
                self.render_front(error = error, title = title, art =art)
```

Now we can submit some art in our browser and see if we get an exception. Even if you don't see any exceptions, there is something you can do in Google App Engine to check that this submitted properly.

You may have noticed when you started up App Engine, that it mentions in the console that the Admin Console is available at a given URL. The Development Console defaults to the Datastore Viewer, and selects all of the entity kinds we have. In this case, we could click List Entities to see all of the entities in Art and check that the new piece of art has map coordinates.

Now that this is all working, we're ready to move on to the next task on our to-do list which is actually drawing the map.

Querying Coordinate Information

The first thing we need to do is to figure out how to draw the map from the API. We know that the URL is going to look like this:

http://maps.googleapis.com/maps/apis/staticmap

followed by some parameters. The only required parameters are:

- size - which is going to be the rectangular dimensions of the map image.
- sensor – this specifies whether the application requesting the static map is using a sensor to determine the user's location. In this case we will set this to False.

Once we have these, we can set the Static map markers parameters. There are a number of optional styles that we could set if we wanted to (size, color, label), but we can just say markers followed by the coordinates. An example URL might look like this:

```
http://maps.googleapis.com/maps/api/staticmap?size=512x512
&markers=40.702147,-74.015794&sensor=false
```

The API works by looking for multiple markers parameters, so we can just add one markers parameter for each of our URLs.

So, there are three steps to generate the image URL:

1. Find which arts have coordinates
2. If we have any arts coordinates, make an image URL
3. Display the image URL

The first step is fairly easy. We just go through our list of arts and for each item check if it has coordinates:

```
point = []
for a in arts:
        if arts.coords:
                points.append(a.coords)
```

An alternative, shorter, method would be:

Points = filter(None, (a.coords for a in arts))

(a.coords for a in arts) is a generator that would return an iterator of all the coordinates, which may be either coordinates or None.

Then filter() takes two parameters. It takes a function or None and a list or an iterable and basically returns all of the items in the list that match the filter.

In this case, if the filter is None, it basically says match all the ones that aren't None and will give us all of the coords for each a in art if it's not None.

There is one subtle bug in our code:

```
def render_front(self, title="", art="", error=""):
        arts = db.GqlQuery("SELECT * "
                           "FROM Art "
                           "WHERE ANCESTOR IS :1 "
                           "ORDER BY created DESC "
                           "LIMIT 10",
                           art_key)

        #find which arts have coords
        point = []
        for a in arts:
                if arts.coords:
                        points.append(a.coords)
        #if we have any arts coords, make an image URL
        #display the image URL

        self.render("front.html", title=title, art=art, error = error, arts = arts)
```

arts is a query. When we iterate over arts, we are actually running the query. We also run the query when we're rendering the front.html. The template has a loop in it that draws all the arts which also iterates over that arts query.

Now, we don't want to run the query twice. Not only is it wasteful, the results of the query could change between the two iterations. Whenever you find yourself iterating over results you get from a database, whether it's datastore or some other kind of curser based abstraction from a database, it's usually good to say something like this:

```
arts = list(arts)
```

The arts that comes out of the query is a curser. What we have done here is to call the list constructor on that curser and say, in effect, make a list out of that iterable. Now we have cached the results of the query in the list and we can iterate over this list as many times as we want.

This is a good habit to get into if you think you might to be using results from a database query more than once.

Quiz: Creating the Image URL

Implement the function gmaps_img(points) that returns the Google maps image for a map with the points passed in. A example valid response looks like this:

http://maps.googleapis.com/maps/api/staticmap?size=380x263&sensor=false&markers=1,2&marker=3,4

Note that you should be able to get the first and second part of an individual Point p with p.lat and p.lon, respectively, based on the above code. For example, points[0].lat would return 1, while points[2].lon would return 6.

Putting It All Together

OK, so now we have a function that, given a list of points, returns the URL for the map image. Let's add the function to the application:

```
GMAPS_URL = "http://maps.googleapis.com/maps/api/staticmap?size=380x263&sensor=false&"
def gmaps_img(points):
        markers = '&'.join('markers=%s,%s' % (p.lat, p.lon) for p in points)
        return GMAPS_URL + markers
```

Now we add the code to make a URL image if we have any coordinates:

```
        #if we have any arts coords, make an image URL
        img_url = None
        if points:
                img_url = gmaps_img(points)
```

Next we are going to pass img_url into our template:

```
        self.render("front.html", title = title, art = art, error = error, arts = arts img_url = img_url)
```

Finally, we have to update the front.html template. We want the image to display to the right of our form, so we add the code in the Jinja template language just after the </form> closing tag:

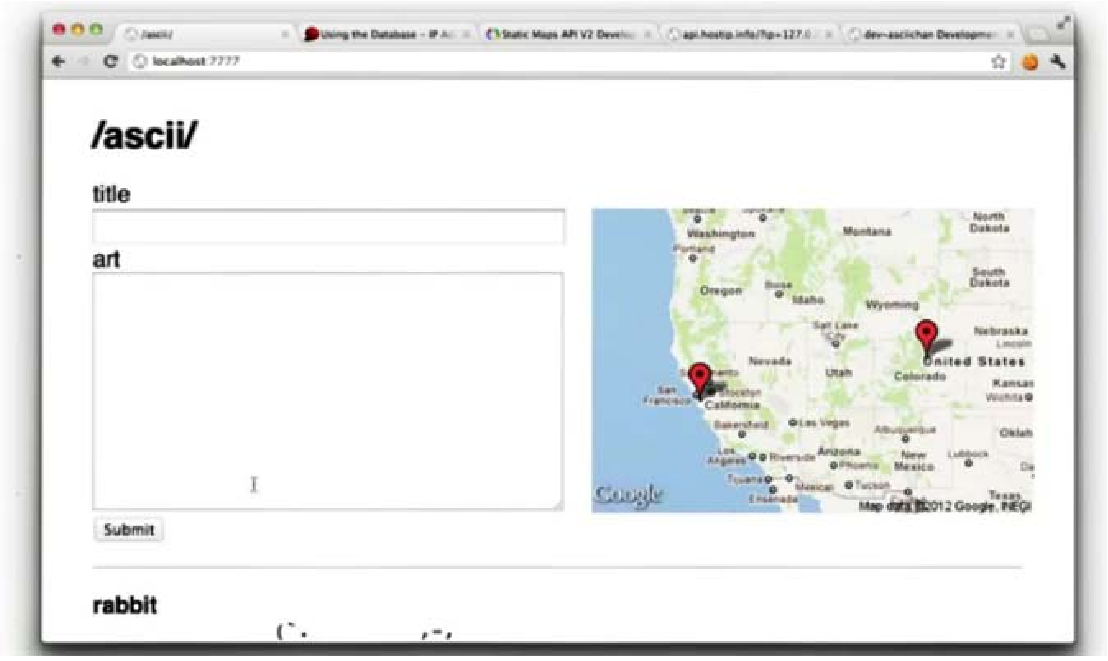
```
        {% if img_url %}
          <img class="map" src="{{img_url}}">
        {% endif %}
```

So, if img_url (i.e. it’s not None), include the image.

Notice the class=”map” attribute. By default, the map would be displayed directly below our form in the browser window. We want it to be displayed to the right of the form. We do this by adding the class attribute and specifying the layout using the CSS styles. To specify the layout we added the following definition to the CSS:

```
.map {
        position: absolute;
        right: 0;
        top: 112px;
}
```

position: absolute allows us to position something anywhere on the page. We have specified that the map should be zero pixels from the right and 112 pixels from the top of the screen. It fits perfectly. Obviously we knew the size of the image beforehand.



The complete ASCIChan code now looks like this:

```
import os
import re
import sys
import urllib2
from xml.dom import minidom
from string import letters
import webapp2
import jinja2
from google.appengine.ext import db

template_dir = os.path.join(os.path.dirname(__file__), 'templates')
jinja_env = jinja2.Environment(loader = jinja2.FileSystemLoader(template_dir), autoescape=True)

art_key = db.Key.from_path('ASCIIChan', 'arts')

def console(s):
	sys.stderr.write('%s\n' % s)

class Handler(webapp2.RequestHandler):
	def write(self, *a, **kw):
		self.response.out.write(*a, **kw)

	def render_str(self, template, **params):
		t = jinja_env.get_template(template)
		return t.render(params)

	def render(self, template, **kw):
		self.write(self.render_str(template, **kw))

GMAPS_URL = "http://maps.googleapis.com/maps/api/staticmap?size=380x263&sensor=false&"
def gmap_img(points):
	markers = '&'.join('markers=%s,%s' % (p.lat, p.lon) for p in points)
	return GMAPS_URL + markers

IP_URL = "http://api.hostip.info/?ip="
def get_coords(ip):
	ip = "4.2.2.2"
	url = IP_URL + ip
	content = None
	try:
		content = urllib2.urlopen(url).read()
	except URLError:
		return

	if content:
		d = minidom.parseString(content)
		coords = d.getElementsByTagName("gml:coordinates")
		if coords and coords[0].childNodes[0].nodeValue:
			lon, lat = coords[0].childNodes[0].nodeValue.split(',')
			return db.GeoPt(lat, lon)

class Art(db.Model):
	title = db.StringProperty(required = True)
	art = db.TextProperty(required = True)
	created = db.DateTimeProperty(auto_now_add = True)
	coords = db.GeoPtProperty( )
```

```
class MainPage(Handler):
    def render_front(self, title="", art="", error=""):
        arts = db.GqlQuery("SELECT * "
                            "FROM Art "
                            "WHERE ANCESTOR IS :1 "
                            "ORDER BY created DESC "
                            "LIMIT 10",
                            art_key)
        #prevent the running of multiple queries
        arts = list(arts)
        #find which arts have coords
        img_url = None
        points = filter(None, (a.coords for a in arts))
        if points:
            img_url = gmap_img(points)
        #display the image URL
        self.render("front.html", title = title, art = art, error = error, arts = arts,
                    img_url = img_url)

    def get(self):
        self.write(self.request.remote_addr)
        self.write(repr(get_coords(self.request.remote_addr)))
        self.render_front()

    def post(self):
        title = self.request.get("title")
        art = self.request.get("art")

        if title and art:
            p = Art(parent=art_key, title = title, art = art)
            #lookup the user's coordinates from their IP
            coords = get_coords(self.request.remote_addr)
            #if we have coordinates, add them to the art
            if coords:
                p.coords = coords

            p.put()

            self.redirect("/")
        else:
            error = "we need both a title and some artwork!"
            self.render_front(error = error, title = title, art =art)

app = webapp2.WSGIApplication([('/', MainPage)],
        debug=True)
```

ASCII Chan is now online at http://www.asciichan.com/

Answers

Quiz: Using urllib

Apache/2.2.3 (CentOS)

```
    >>> p2 = urllib2.urlopen("http://www.example.com")
    >>> p2.headers['server']
    'Apache/2.2.3 (CentOS)'
```

Quiz: XML and HTML

- All HTML is XML?
- All XML is HTML?
- **HTML can be expressed in XML?**
- **XML and HTML share a common lineage.**

Quiz: Parsing RSS

12 (at the time of writing)

```
>>> import urllib2
>>> from xml.dom import minidom
>>> p = urllib2.urlopen("http://www.nytimes.com/services/xml/rss/nyt/GlobalHome.xml")
>>> c = p.read()
>>> x = minidom.parseString(c)
>>> x.getElementsByTagName("item")
[<DOM Element: item at 0x2899df0>,
<DOM Element: item at 0x28a40f8>,
<DOM Element: item at 0x28a8058>,
<DOM Element: item at 0x28ae710>,
<DOM Element: item at 0x28b4b48>,
<DOM Element: item at 0x28b8da0>,
<DOM Element: item at 0x28f1738>,
<DOM Element: item at 0x28f7cb0>,
<DOM Element: item at 0x28fdd00>,
<DOM Element: item at 0x2902aa8>,
<DOM Element: item at 0x290c300>,
<DOM Element: item at 0x2912300>]
>>> len (x.getElementsByTagName("item"))
12
```

Quiz: Parsing JSON

103978

```
def total_ups():
    front_page = json.loads(reddit_front)
    return sum(c['data']['ups'] for c in front_page['data']['children'])
```

Quiz: Escaping JSON in Python

```
{"blah":["one", 2, "th\"r\"ee"]}
```

Quiz: Good Habits

Which of these are good habits to get into?

- **Sending proper user agents?**
- Writing custom protocols?
- Using SOAP?
- **Rate-limiting yourself?**
- **Using common protocols and data formats?**

Quiz: Geolocation

```
from xml.dom import minidom

def get_coords(xml):
    p = minidom.parseString(xml)
    long_lat =p.getElementsByTagName("gml:coordinates")
    if long_lat and long_lat[0].childNodes[0].nodeValue:
        lon, lat = long_lat[0].childNodes[0].nodeValue.split(',')
        return lat, lon
```

Quiz: Creating the Image URL

```
from collections import namedtuple

# make a basic Point class
Point = namedtuple('Point', ["lat", "lon"])
points = [Point(1,2),
          Point(3,4),
          Point(5,6)]

GMAPS_URL = "http://maps.googleapis.com/maps/api/staticmap?size=380x263&sensor=false&"

def gmaps_img(points):
    markers = '&'.join('markers=%s,%s' % (p.lat, p.lon) for p in points)
    return GMAPS_URL + markers

print gmaps_img(points)
```