HTTP Requests

Review of how websites work

The basic way things work when you want to access a website is your web browser sends an HTTP (*hypertext transfer protocol*) request over to the web server where the website is hosted, asking for a particular file. The server will then send back that file. The file might already exist, but it might not. Often the server will generate the page on the fly. For example, when a student views their grades in their university's online system, there is not a specific page that already exists with those grades. Instead, the server reads the grades from a database and generates a page to send back.

The HTTP request sent over to the browser follows a particular format that will be described in detail later. It includes what page is being looked for, what version of HTTP to use, various headers specifying information about the client, and possibly some data. The server responds with a status code indicating whether the request was successful or not, as well as various headers giving information about the server and what it is returning. The actual page requested is at the end of the response.

Usually a visit to a website generates several HTTP requests, often dozens, looking for various resources like the page content, image files, and scripts. Most webpages are HTML files. Those files usually contain CSS and JavaScript as well. Your browser reads the HTML code of a page and uses that to format how the page looks. It uses CSS to refine the formatting, often for things like font sizes and coloring. JavaScript is the main programming language that your browser understands. It's used to add interactivity to web sites. Without it, sites would mostly just be static documents, not all that different from PDFs or Word docs.

HTML is a markup language which uses tags to specify how things will be displayed. For instance, here is a little HTML:

```
<ul>
    <li> This is an item in an unordered list</li>
    <li> Here is a link: <a href='http://example.com'>click here</a></li>
</ul>
```

Usually there is an opening tag and a closing tag. For instance, `<ul>` starts an unordered list and </ul> ends it. Another example is the <a> tag, which creates a link. Here is an example:

```
See <a href="http://www.example.com">this site</a> for more information.
```

The link's target is given by `href` and the text that the user can click on to follow the link comes between the starting tag and the ending </a> tag. You can learn the basics of how HTML works pretty quickly. CSS is primarily used for specifying how a website is formatted. Here is a short example:

```
a:hover {
    background-color: #DD0;
    color : black;
}
body {
    margin-top : 20px;
    margin-left : 20%;
}
```

We won't need to know much about CSS other than just to be able to recognize it. JavaScript is a language that looks a lot like Java, though it's different. JavaScript is what allows for most user interaction with a site. JavaScript can be included in a page by putting it between HTML `<script>` tags. You can also use the script tag to load an external JavaScript file, including one from a different site. Many website vulnerabilities come from JavaScript. Here is a little JavaScript:

```
<script>
var input = document.getElementById("inputBox");
var button = document.getElementById("convertButton");
var div = document.getElementById("output");
function buttonClicked(e) {
    var temp = parseFloat(input.value);
    temp = 5/9*(temp - 32) + 273.15;
    div.innerHTML = Math.round(temp*10)/10;
}
button.addEventListener('click', buttonClicked, false);
</script>
```

The client's web browser is mostly limited to HTML, CSS, JavaScript, and one other thing, WebAssembly, which we won't cover here. On the server, however, there are no limitations. You can run whatever language you want there and use it to generate the pages that are sent back to the client. Common server-side languages include PHP, JavaScript, Python, Java, and Ruby.

Sending data to a website

Here is a simple HTTP request:

```
GET /stuff/somefile.html HTTP/1.1
Host: example.com
```

The first line starts the command type, which is often GET or POST. More on these in just a bit. Next comes a path to the file being requested. The last part indicates what version of HTTP is being used. The second line is an example of a *header*. The host header tells the web server which specific site you're looking for in case it's hosting multiple sites. There are usually other headers, but we haven't included them here. The web server will then send the file you requested. That response contains various headers indicating things like dates, file sizes, and it contains the file you asked for.

The two common ways to send data are via HTTP GET requests and HTTP POST requests. The difference is that GET sends the data in the URL, while POST sends it in the body of the HTTP request. In your web browser, usually this is done via a form. Here is an example form in HTML:

```
<form action="form.php" method="get">
    Enter your age: <input type="text" name="age" />
</form>
```

This uses GET and will send the info you put into the form to a URL like `http://example.com/form.php?age=25`. If the form had another field called `class`, the URL might end up looking like this: `http://example.com/form.php?age=25&class=senior`. The part that starts at the question mark is called a *query string*. It consists of name=value pairs with different pairs separated by `&` symbols. Certain special characters, like the `/` character need to be encoded. This is called *URL encoding*, and it consists of a `%` symbol followed by the ASCII value of the character in hex. For instance, `/` is `%2f`.[1]

If this were a POST request, the `method` field above would be `"post"` and the information would be sent in an HTTP request that looks like this:

```
POST /form.php HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 20

age=27&class=senior
```

[1]It is also possible to URL-encode Unicode characters, though we won't cover that here.

For GET requests, the data is always visible in the URL. For POST requests, the data is not in the URL. For both types of requests, you can use your browser's developer tools to see the info your browser is submitting to a site. You don't have to use your browser to submit data to a page. There are several other ways.

1. For GET requests, you can just craft the URL. For instance, for the form above, you could set the age directly in the URL, and that will bypass the form entirely.
2. For both GET and POST requests, you can send the data directly in an HTTP request like the ones above using a tool like Telnet or Putty.
3. Most programming languages have libraries that let you send HTTP requests directly. Python has a nice one called `requests`. It doesn't come in the standard Python distribution, so you have to install it separately.[1] Here is a simple example that does both a GET and a POST request:

   ```
   page = requests.get('http://example.com')
   page2 = requests.post('http://example.com/form.php', {'price':-50, 'name':'chocolate'})
   print(page.text)
   ```

   This prints out just the text of the page returned. It's also possible to see status codes, response headers, and much more, as well as specify request headers. Note in the post request that the data to be sent is put into a dictionary. Note that if you try to run this exact command, it won't work since there is no such page at `example.com`.
4. There are command line tools that can be used to send HTTP requests. The most popular one is called *cURL*. It comes on Mac and Linux, but not Windows, so if you want to use it on the Windows command line, either download curl or use PowerShell's Invoke-RestMethod. Here is an example of using cURL to send some data to a (nonexistent) form at `example.com`:

   ```
   curl -d "age=25" http://example.com/form.php
   ```
5. There are special tools like Burp Suite that allow you to edit all sorts of things you send to a site in a browser environment.

Here is some code from a form. It tries to use JavaScript to prevent people from entering a negative price. However, by sending our own POST request using any of the methods described above, we can totally bypass the script (which only runs in the browser) and set a negative price. In order to prevent a negative price, a client-side check in JavaScript is not enough since that check will only run in a browser. Instead, the check needs to be done on the server. People using the form have control over the data sent to the server, but they don't have control over the code that runs on the server.

```
<form action="form.php" method="post" id="whatever">
        Price: <input type="text" name="price" id="priceEntry" /><br />
</form>

<script>
function func(e) {
    var entry = document.getElementById("priceEntry");
    var value = parseInt(entry.value);
    if (value < 0) {
        e.preventDefault();
        alert("Invalid price");
    }
}

var form = document.getElementById("whatever");
form.addEventListener("submit", func);
</script>
```

[1] You can install it at the command line using `pip install requests`. On a Mac, if you want to be sure you're installing it for Python3, use `pip3` instead of `pip`. On Windows, depending on whether or not Python is in your path, you might need to navigate to the `Scripts` folder of your Python installation before calling `pip`. You can find the folder Python is installed in by importing the `os` library and calling `os.getcwd()` at the Python shell.

Headers

There are a couple of dozen headers that can be set in HTTP requests. Usually your web browser is responsible for setting them. They can specify things to do with language, caching, asking for specific bytes of a file, and more. Some important request headers are listed below.

- `Cookie` — Used for cookies; more about this later.
- `Host` — As mentioned earlier, often multiple sites are hosted at the same IP address; this is used to tell the server which one you want.
- `Referer` — Note that it's actually misspelled like this in the HTTP spec (proper spelling is "referrer"). This header is used for your browser to indicate what page it was at when the user clicked a link to get to the resource being requested. For instance, if you search Google for Mount St. Mary's and click on the link to the Mount's homepage, the referer header in the request to `msmary.edu` will be set to `google.com`. The referer header is one way for the operator of a web site to know how people are getting to their site.
- `User-Agent` — This is a string that usually gives information about the web browser making the request. The string is long and convoluted for historical reasons, but it usually contains the browser type and operating system info of the client. The server can use the user agent string decide what to send back based on the browser type. Sometimes servers will ignore requests if the user agent string does not match something that looks like a real browser.

There are also a few dozen response headers. Some have to do with caching, some tell about the file type and encoding. One header identifies the type of web server. There are also a response header for cookies.

Status codes

Status codes are three-digit numbers that indicate what happened with the request. The most well known code is 404, for page not found, which we've all seen many times. Status codes follow this scheme:

- 100s — Informational message (not used very often)
- 200s — Success (to say that the page was successfully sent)
- 300s — Redirect (to indicate page has moved or is located elsewhere)
- 400s — User error (the person downloading the page has done something wrong)
- 500s — Server error (something went wrong on the server)

There are several dozen status codes in total. Here are a few of the more common ones:

- 200 OK — This is what you usually get if everything went okay.
- 301 Moved Permanently — This is used to tell the user/web browser that the page is no longer at this location and to give its new location.
- 304 Not Modified — Used for caching. Specifically, the server can send this status code to let the client know that there is not a newer version of the resource than what the client has in their cache.
- 400 Bad Request — You'll see this if you're experimenting with manual HTTP requests and you do something wrong.
- 403 Forbidden — If a page requires authorization and you don't provide it, you may get this message. Often you'll see this is you try to look at a directory listing, often by editing a URL.

- 404 Not Found — The page you're looking for either doesn't exist, or maybe it did and was deleted and now you're out of luck.
- 500 Internal Server Error — Generic error message for a problem on the server.
- 503 Service Unavailable — Sometimes you'll get this if a site is overloaded with traffic.

URLs

A URL is a *uniform resource locator*. Examples are `http://www.example.com` and `https://brianheinold.net/python/python_book.html`. Most URLs follow the description below. If you want to see all the intricate details, see the Wikipedia page on URLs.

1. URLs start with a scheme, which is usually `http` or `https`, though there are other possibilities, like `file` or `ftp`.
2. Next comes the host name, which is usually a domain name, such as `www.example.com`. After the host name you can optionally put a colon and a port number, like `example.com:8080`. This is used if you want to access a page at a nonstandard port. Usually it's left out, but sometimes people put their sites at unusual port numbers. This can be to hide the site or because something else if running at the standard port.
3. Next comes a path indicating directories and the file you want. In `https://brianheinold.net/python/python_book.html`, the path is `/python/python_book.html`, indicating the directory and file name of the desired resource. It's often possible to leave off the file name. This will either allow you to view the directory contents or go to a default file in that directory.
4. Next comes the query string. This is optional. It starts with a question mark and contains name/value pairs separated by ampersands. This is used to send info to the server. A typical example might look like this:

   ```
   http://example.com/form.php?name=steve&age=27&search=chocolate
   ```

   Query strings are used as a way to send data to a server in the URL. Many forms use this. For example if you do a search at Duck Duck Go for wikipedia, the URL generated is the following:

   ```
   https://duckduckgo.com/?q=wikipedia&t=hk&ia=web
   ```

   The `q=wikipedia` is generated based on what we put into the form at Duck Duck Go. If you change the URL in the URL bar to `q=computers`, it will take you to the search results for computers. (The other two things in the query string are things Duck Duck Go adds for reasons we won't worry about here.)
5. Finally comes the fragment portion. It is an optional link to a specific portion of a page. It starts with a `#` symbol. For instance, `https://en.wikipedia.org/wiki/Url#History` links directly to the history part of the Wikipedia article on URLs.

URLs are a special case of something called URIs (uniform resource indicators). Some pedantic people get angry if you mix them up, but many people use the terms interchangeably.

URL encoding Certain characters in URLs, like the question mark, have special meanings. If we need a question mark at a certain place in our URL, we have to *escape* it. This is done by using the `%` symbol followed by the symbol's ASCII or Unicode character code in hex. The code of the question mark is 3F in hex, so it would be encoded as `%3F`. If you go to Duck Duck Go and search "what is http?", the query string it generates will look like this:

```
https://duckduckgo.com/?q=what+is+http%3F&t=hk&ia=web
```

We see that the question mark in our query is replaced with `%3F`. Notice also that the spaces of our search are replaced with plus signs. This is because spaces are not allowed in URLs, and the rule is to replace them with plus signs.

As another example, if we enter `abc123!@#$%^` into a form, it will be transformed into `abc123!%40%23%24%25^`. Notice that some of the special characters are encoded, while others aren't. There are rules specifying which ones must be encoded, but the details aren't important here. Note also that any character can be URL-encoded. For instance, `%61` is the encoding for a lowercase *a*. Sometimes people use this trick to get around security precautions. If a site prevents you from entering the word `cab` into a form, it's possible you could get around that by putting `%63%61%62` into the query string.

Cookies

Cookies are a way for a web server to store a small amount of data on a client's device.

The way cookies work is as follows: When a client first visits a web site, that web site sends back a Set-Cookie response header which contains the name and value of the cookie. Along with those there is also an expiration date for the cookie. The web browser stores all that info. Then every time a client visits the site again (until the cookie expires or is deleted), their browser will send back the cookie name and value to the server in the Cookie request header. The key point to remember is that the cookie is sent back to the server with *every* request.

You can use your browser to view all the cookies it stores for a particular site. On Chrome, you can do this by clicking on the icon to the left of the domain name in the URL bar. In Firefox, they are in the web developer tools under the storage tab. You can also use your browser to delete cookies.

Here are some common applications of cookies.

1. A simple application of cookies is for a web page to remember user settings. Whenever the user visits the site, their browser will send a cookie containing the user's settings. The web page can then send back appropriate data based on those settings.

2. One of the most common applications is for logging onto a page. When you first log on with a username and password, the server sends back a session ID. This is a long, random string that is unique to you. Every time you make a request to that page, your browser sends the server a cookie containing the session ID. That's how the server recognizes it's you. It's important to make sure no one else obtains that session ID because they could use it to impersonate you.

3. An unfortunate use of cookies is for tracking which websites people visit. This is done usually through *third-party cookies*. When you load a webpage, that page often makes requests to a variety of different domains. Some of those domains hosts resources, like images, that the page needs. Others host JavaScript files used by the page. Each time you load one of those resources, your browser sends any cookies set by the domains of those resources to those domains.

   Here then is how the tracking happens. A tracking network has to find a way to get one of their resources on a variety of different sites. Sometimes they have agreements with those sites. They could also buy ad space on those sites. Often these resources are tiny 1-pixel images that you would never notice. The tracking network makes sure the name of the resource is specific to the site. Then when you visit the site, your browser will load the tracking network's resource and send any cookies for the tracking network's domain back to that domain. The tracking networks make sure the values of those cookies are identifiers that are unique to each client. Combining those identifiers with identifiers of which page the resource is on allows the tracking network to build a dossier of which sites you visit. They could use this to show targeted ads to you. They could also sell that dossier of sites you visit to various unsavory parties.

Brute-forcing forms

Sometimes you want to try a bunch of different inputs at a form, and it would be too tedious to try to enter them all in by hand. Below is hypothetical example using Python's requests library where we want to send a bunch of dictionary words to a form until we see the word “Success” show up on the page.

```python
import requests
from time import sleep

words = [line.strip() for line in open('wordlist.txt')]
for w in words:
    t = requests.post('http://example.com/someform.php', {'data':w})
    if 'Success' not in t.text:
        print(t.text, w)
        break
    sleep(.1)
```

The call to the `sleep` function is important. If you make too many requests to a site in too short of a time, they may decide to block your IP address. Including the sleep call is also a nice thing to do to prevent their server from getting overwhelmed.

A related concept to this is *web scraping*, where instead of sending data to a page like this, we automate a bunch of requests to different pages on the site, often if we want to download a bunch of data from the site. You can either write your own scrapers using code a little like the above, or use a library already built for that purpose.

Web drivers

Many pages don’t take GET or POST data, and the only way to interact with them is with a browser using JavaScript. If we want to automate the process of trying out lots of inputs on the page, the approaches discussed earlier won’t work. Instead, the proper tool is a *web driver*. This is a program that you can use to automate interactions with a web browser.

Probably the most popular one right now is the Selenium web driver. To install it, you’ll need to download it using your favorite programming language’s package manager. In Python, this would be `pip install selenium` at the command line. You’ll also need to install the web driver executable. Do a search online for it. For Chrome, it’s at `http://chromedriver.chromium.org/downloads`. Make sure to download the version number that fits with the version number of you browser, and also make sure that the executable is either in your path or in the same directory as the code you write. Here is a little code that sends input to a simple JavaScript temperature converter.

```python
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get('http://example.com/somepagethatdoesntreallyexist.html')

driver.find_element(By.ID, 'inputBox').send_keys('32')
driver.find_element(By.ID, 'convertButton').click()
```

The URL is to a nonexistent page, but assume that the page has the following HTML code:

```html
<input id="inputBox" type='text'/>
<button id="convertButton">Convert</button>
```

The Python code uses the `driver.find_element` function to get references to the elements on the page. When working with web drivers, this is probably the trickiest part of things. You’ll have to go spelunking through the page source to figure out how to access the elements on the page. See the Selenium documentation online for all the ways to access elements and how to simulate user interaction.

XHR requests

XHR is short for *XML HTTP request*. They are also called AJAX requests (*asynchronous JavaScript and XML*). The main idea of these is that information can be sent to a server without the page having to be reloaded. The server will often send information back in response to one of these requests. Because there is no page reload, you might not even notice that one of these requests is happening in the background. For instance, a page can tie an event listener to keypresses and send every keypress you make on the page over to the server. The following JavaScript code does just that:

```
var xhr = new XMLHttpRequest();
document.addEventListener('keyup', makeRequest, false);
function makeRequest(e) {
    xhr.open('POST', 'serverpagethatcollectskeystrokes.php');
    xhr.setRequestHeader('Content-type", "application/x-www-form-urlencoded');
    xhr.send('key=' + e.code);
}
```

To see if a page is making XHR requests, go the Network tab in your browser's developer tools. There is usually a tab to specifically show XHR requests.

Browser developer tools

As noted a few times earlier in this set of notes, web browsers come with an extensive set of developer tools. You can usually access them by right-clicking on the page and choosing `inspect element` or something similar. You can also access them through the browser's menu system. You can use those tools to many useful things. Here are a few:

- View all the elements of the HTTP requests made by your browser,
- View, modify, and delete cookie values
- View and modify the source of the page (which will only affect your copy of it, not anything on the server)
- Access the JavaScript console to view and change the values of JavaScript variables.

Caching

Recall that *caching* is a computer science term for storing data. In HTTP, your browser and other machines can cache frequently accessed pages to save the time and network usage of constantly downloading the page from the server. There are various request and response headers to help with this. The basic idea is that when you download a page, your browser may save a copy of it locally. When you go to download the page again, your browser will send an HTTP request with some of the caching headers set basically asking the server if it has a newer copy of the page than the one you downloaded. If it does, it will send it to you, and if not, it will send back an HTTP 304 Not Modified status code.

Proxy servers

In English, the word *proxy* means someone that takes someone else's place. For instance, a person who is supposed to attend a meeting but can't make it might send a proxy in their place. In the internet, proxy servers stand between the client and the web server. There are proxies that stand in for clients and there are proxies that stand in for web servers.

Client-side proxies have a few different roles. Any request you send will pass through that proxy before it goes out to the internet. That proxy could be used to block certain requests, possibly to stop people on a network from accessing certain pages. That proxy can also be used to view encrypted communications. Instead of having a direct connection to an outside web server, you might have a connection with the proxy who then has a connection to the server on your behalf. Because there's not a direct encrypted connection between you and the server, the proxy will be able to read the data sent back and forth.

Client-side proxies are also used by people to hide their IP addresses from web servers. In order to make an HTTP request, you need an IP address, and web servers usually log that. If you want to protect your privacy or if you want to get around IP address restrictions, you could use a proxy. Note, however, that the proxy itself will see your IP address and all the pages you are requesting, so you have to trust it.

Finally, client proxies can also be used as caches. If there are several clients behind the proxy that all want the same page, the proxy can download it just once itself and serve cached copies of it to all the clients, saving some network usage.

Server-side proxies (often called *reverse proxies*) sit in front of a web server. One use is for security, adding an additional layer between the internet and the web server. The proxy could be used to filter out certain types of requests to prevent them from getting to the web server. A common use is for DDoS protection. Server-side proxies can also be used for load-balancing. Big sites tend to have multiple web servers. A proxy could be used to spread out the requests evenly among the web servers.

JSON

JavaScript Object Notation (JSON) is a very common way of transferring data over the internet. It's especially useful for data in which there are several fields. Storing the data in a standard format like JSON allows us to easily parse the data, especially since there are libraries built into most programming languages for that purpose. Here is a sample JSON document.

```
{"talks":[
        {"date":"September 18, 1992",
         "name":"Carrie Oakey ",
         "title":"How to sing badly"},

        {"date":"October 26, 1983",
         "name":"Cory Ander",
         "title":"Cooking with spices"},
]}
```

Here is some Python code that will parse through this for us. Assume that the data is located in a file called `talk_data.json`.

```python
import json
parsed = json.loads(open('talk_data.json').read())

for x in parsed['talks']:
    print(x['name'])
```

The first line after the import reads the file into a string, which then is passed into the `json.load` function. If we already have the JSON in a string, then we wouldn't need the `open` line. The loop reads items in the list `parsed['talks']`. The `talks` part of it comes from the fact that the JSON data we are looking at contains a list called `talks`. We can then access individual fields in each item using things like `x['name']`, `x['date']`, etc.

XML

Extensible Markup Language (XML) is another way of structuring data. It plays a similar role to JSON. Some internet services use XML, while others use JSON. Both are pretty common. XML's syntax was inspired by HTML. Various fields are started with a tag, like `<date>` in the example below and the end of the field is marked by a corresponding closing tag, like `</date>` in the example below.

```
<?xml version="1.0" encoding="UTF-8"?>
<talk_information>
    <talk>
        <date>September 18, 1192</date>
        <name>Carrie Oakey</name>
        <title>How to sing badly</title>
    </talk>

    <talk>
        <date>October 26, 1983</date>
        <name>Cory Ander</name>
        <title>Cooking with spices</title>
    </talk>
</talk_information>
```

Here is some Python code to parse this XML.

```python
import xml.etree.ElementTree as ET
tree = ET.parse('smalltalk.xml')

for x in tree.getroot():
    print(x.find('name').text)
```

Notice that there are some differences to how the XML parser works versus how the JSON parser works. But the overall approach is similar.