HTTP

The *Hypertext Transfer Protocol* (HTTP) is the protocol behind transferring web pages. When a client (like your web browser) wants to request a page from a server, it sends that request in the format specified by HTTP. This is called an *HTTP request.* The server returns the page you asked for along with some other details about the page. This is called an *HTTP response.*

Here is the simplest possible HTTP request:

```
GET /
```

You can try this out using telnet (Mac/Linux) or Putty (Windows). Specifically, connect to web server, such as `www.example.com` at port 80, and type in the request above. On a Mac, you will probably need to install telnet. In Putty, you will need to select "raw" for the connection type and tell it to not to close the window on exit. If you do things right, you will get a response back. It's pretty neat to be able to download a web page by making a direct connection yourself without having to rely on a web browser.

The HTTP request above is the original way of doing things, going back to HTTP version 1.0. Some web servers will answer requests like that, but many will send back an error message. The following is a more modern type of request that works with most web servers. In it, replace `example.com` with the host name of whatever site you are accessing.

```
GET / HTTP/1.1
Host: example.com
```

In general, an HTTP request follows a format like above. The first line is in the form below:

¡command¿ ¡resource location¿ HTTP/1.1

The most common command types are GET and POST. More on them later. Two other commands, PUT and DELETE, are for uploading and deleting files on the server. If you're running a web server, you will usually want to make sure those commands are disabled so that people don't delete things from your server or store things you don't want there. One other useful command is HEAD, which returns header information about the page without returning the page itself. This is useful before you download a page if you want to know info like how large the page is.

The resource location is the name of the page you want on the server. Putting a single slash asks for the default homepage. If you wanted a file `passwords.html` in the directory `admin`, then you would use `/admin/passwords.html`. The last part of the first line indicates the version of HTTP being used. Right now 1.1 is the most common one.

After the first line of the HTTP request there are usually several header lines. These are ways of sending additional info about your request to the server. The one in the example above is called `Host`, and it's required in HTTP version 1.1. The reason for it is that some sites host multiple web sites at the same page and they use the `Host` header to tell which site is being asked for.

The server will respond with a status code, which tells to what degree the server was able to (or not) serve the request. After that there are response headers that give information about the page being requested. After the headers usually comes the actual contents of the page.

The telnet/Putty example is one way to view the contents of an HTTP request. A more practical way is to use the developer tools of your web browser. To do this, open up the developer tools before loading the page. Then go to the network tab and load the page. You will be able to see all of the request and response headers, as well as the status code.

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. The most important request headers are listed below.

- `Connection` — It's usually set to either `close` or `keep-alive`. In HTTP 1.0, connections were always closed after a page was done downloading. However, in the modern internet, often when you load a page, dozens of resources, like images and scripts, are loaded. Setting up a new connection for each individual resource would be very slow, so `keep-alive` is used to tell the server to reuse the connection for other resources instead of starting up new connections.
- `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.

GET vs POST

In HTTP requests, the two most common commands are GET and POST. Both can be used to request a web page from a server, and both also can be used to send data to the server in the request. The difference is that GET requests send the data in the URL query string, while POST requests send the data inside the body of the HTTP request. This makes POST requests somewhat more secure than GET requests, as the data is not in plain site in the URL. However, anyone who intercepts a POST request can still easily view the data, so POST by itself is not enough to keep data secure. Some type of encryption is also needed.

Since GET requests store the data in the URL, they are useful for things that you would want to bookmark. For instance, the search terms from a Google search are sent in a GET request. This allows you to bookmark the page of results. On the other hand, when you order something online, the data is usually sent in a POST request. The idea here is that the addition to the shopping cart is not something you want to be long-lived. It happens once and then it's done.

Let's look at an actual GET HTTP request versus a POST request. Suppose we are submitting data to a form at `example.com/form.php`. There are two fields, `name` and `age`, that we are setting to the values `steve` and 27. Here's what a GET request for this looks like. Notice that the data is sent in the URL.

```
GET /form.php?name=steve&age=27 HTTP/1.1
Host: example.com
```

Here is what the equivalent POST request looks like:

```
POST /form.php HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 17

name=steve&age=27
```

Notice that the data is sent in the body of the HTTP request. In order to use POST, a few other headers need to be set to tell the server what format the data is in (URL-encoded in this case) and how long the data is (17 bytes here).

If you visit a page and want to see what data your browser is sending via GET or POST, you can do it with the developer tools under the network tab.

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.

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.

HTTP/2 and HTTP/3

For a long time, HTTP/1.1 was the standard. Then HTTP/2 was introduced in 2015. The major difference between HTTP/2 and HTTP/1.1 is in how connections are handled. Most modern web pages involve dozens of images, scripts, and other resources being loaded. In HTTP/1.0, as we covered earlier, each of those would require a new connection, and creating new connections is slow. HTTP/1.1 allowed the same connection to be reused for multiple resources, allowing for a substantial speedup. But still only one request at a time could be done on that connection. One solution to that is to open multiple connections in parallel, but this suffers from the fact that opening connections is slow. HTTP/2 fixes this by allowing multiple HTTP requests to concurrently use the same connection. Rather than one HTTP request finishing and another starting on that same connection, parts of various HTTP requests are all coming over at the same time.

There is still a weakness to this approach, which has to do with the fact that the connection is a TCP connection. TCP is a protocol we will cover a bit later. The main thing to know about it for the moment is that if a network problem causes some of the packets to be lost, then the whole connection will essentially pause until those packets are resent and received. Even if the packets were only part of one particular HTTP request, this will pause transmission on the entire connection for all the requests. A new standard, HTTP/3, addresses this by replacing TCP with a new protocol called QUIC that essentially keeps the HTTP requests separate on that connection so that if one of them has a problem, it doesn't block the others. As of late 2020, HTTP/3 is in development and is supported major browsers.

Some other differences of HTTP/2 and HTTP/3 include the fact that they are binary protocols as opposed to the plain text HTTP/1.1. Also, they are used exclusively over HTTPs, not plain HTTP. In other words, those connections are always encrypted.