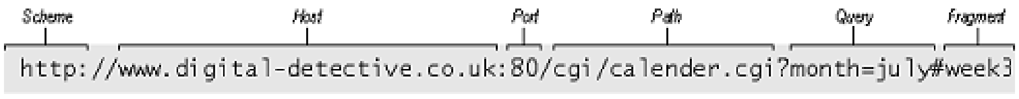

Scheme
Host
Port
Path
Query
Fragment
http://www.digital-detective.co.uk:80/cgi/calender.cgi?month=july#week3

Figure 1: URL structure

HTTP

HTTP is HyperText Transport Protocol. A *protocol* is a language or format for interoperation between different systems. *HyperText* refers to any text with links in it.

URI/URL/URN is a Uniform Resource Indicator/Locator/Name – an identifier for some digital object (text, image, PDF, article, audio/video, etc.) I don't care so much about the distinction between these three, but this is one aspect of it:

An identifier (URI/URN) does not necessarily tell you where to get it from (Ex. book ISBN), but a URL also tells how to access/retrieve the resource (Ex. FTP = file transfer protocol)

Anatomy of a URL

Another example: `http://www.jobs.cam.ac.uk:80/job/5774/`

The part up to the colon is the **scheme,** and is usually `http` but can be `https` (secure), `ftp` (file transfer), `tel` (telephone number), and others. An example telephone URL to my office phone is `tel:17184881274` – maybe that will be clickable when this page is viewed on your mobile phone.

The **host** part of the URL can be an IP address or dotted host name (which can be converted to an IP address by DNS). The **port** can be specified after a colon, but the default port is 80 if it is omitted (actually, 80 for `http` and 443 for `https`). A computer on the Internet can be listening for connections on multiple ports, and use the port number of the connection to determine what services to provide.

The **path** starts with the slash, after the host portion, so `/cgi/calendar.cgi` in the example above. Sometimes it ends with a file-name extension (`.html`, `.pdf`, `.jpg`), sometimes it ends with a slash, and sometimes it's bare.

The **query** starts with a question mark, and consists of a set of parameters (variables) and their values. Multiple parameters are separated with & (ampersand).

Another example: `https://www.youtube.com/watch?v=kGOpY2J31pI&t=0m28s` In this URL, the scheme is `https`, the host is `www.youtube.com`, the port defaults to 443, the path is `/watch` and the query is `?v=kGOpY2J31pI&t=0m28s`, which defines two variables: the video identifier `v=kGOpY2J31pI` and the time at which to begin video playback `t=0m28s`.

Many sites also support short/abbreviated URLs, such as `http://youtu.be/kGOpY2J31pI` for the same video. In this case, it uses the `youtu` domain name within Belgium's `.be` country code.

Another example where you see a query is when you do a Google search. Search for "web jobs" and you'll see `?q=web+jobs` appear in the URL. The plus sign appears because a URL **cannot** have a space in it. It also cannot contain plenty of other special characters, and characters such as =,& are reserved for delimiting query variables, so there are other ways to encode special characters that we'll see later. (Search google for "the & symbol" and notice `?q=the+%26+symbol` appear in the URL.)

GET request

An HTTP conversation is a series of requests/responses. Request always starts from the client, and the server responds. A request begins with one line which has three parts:

- **method** = a verb, what action are we doing. The most common action is to retrieve something, which is called `GET`. We'll see the others later.
- The **path** for this request, which comes from the URL. For example, `/watch?v=kGOpY2J31pI`
- The protocol **version,** such as `HTTP/1.0`

So a complete, correct request line would be

```
GET /watch?v=kGOpY2J31pI HTTP/1.0
```

Following that one line, the client will specify a series of **headers** that modify the request in some way. It's just a generic way for the client/server to exchange further information. Some common examples:

- `User-Agent: Mozilla Firefox (Win8; v35)` identifies what browser or other client software is being used.
- `Accept-Language: da, en-gb, es-ec` identifies the preferred languages of the user, in order. If the web site is multi-lingual, it will try to match its content to the preferred languages. These codes mean Danish, British English, and Ecuadorian Spanish. See `http://www.metamodpro.com/browser-language-codes` for more.
- `Host: www.youtube.com` gives the *host* portion of the URL. If specified, it allows the same web server to serve multiple web sites, called virtual hosting. (Required in version 1.1 of HTTP.)
- `Referer: https://liucs.net/cs120s15/` (yes the name of this header is misspelled!) gives the URL of the page on which this link was clicked (in other words, the previous page in your browser's history).

The headers end with a blank line. So here is a complete HTTP request that my browser used to fetch the language-codes page:

```
GET /browser-language-codes HTTP/1.1
Host: www.metamodpro.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:35.0) Gecko/20100101 Firefox/35.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=
    0CCAQFjAA&url=http%3A%2F%2Fwww.metamodpro.com%2Fbrowser-language-codes&e
    i=uJXCVOOPL4LYggTS_4DQDQ&usg=AFQjCNE2XZTwYEgvHFkw0JQ-vHc2z4ukKA&sig2=h1Q
    SWPH5pxjG9i1JanilBw&bvm=bv.84349003,d.eXY
Connection: keep-alive
If-Modified-Since: Fri, 23 Jan 2015 18:41:00 GMT
Cache-Control: max-age=0
```

All the web standards we're describing are hosted at http://www.w3.org/

POST and other requests

The GET method, described in the previous section, is meant to **fetch** a resource. HTTP is designed so that GET requests can safely be cached and repeated whenever necessary. Therefore, it would be disastrous if a GET were to trigger actions that have important consequences like adding an item to a database or charging a credit card. (There are some horror stories of this happening in the early days of the Web, before HTTP was widely understood.)

So instead of GET, we can use the POST method. Browsers, servers, and proxies know that it would be inappropriate to repeat or cache the results of a POST. When you submit a form on the web, such as to log in to a server, register a new user account, upload a file, or enter your credit card details – those forms are transmitted using POST.

In addition to the headers of the GET request illustrated above, a POST often carries a **payload** (also called the **request body**). This is an encoding of the fields in the form, or of the file being uploaded. In the request below, notice the new headers Content-Length and Content-Type.

```
POST /checkin HTTP/1.1
User-Agent: curl/7.40.0
Host: localhost:3000
Accept: */*
Content-Length: 19
Content-Type: application/x-www-form-urlencoded

name=Chris&score=32
```

When there is a non-zero content length, then the blank line ends the headers section and the server waits for (in this example) 19 additional bytes of payload. The payload shown above looks a lot like the **query string** part of a URL. That format is called `x-www-form-urlencoded`, but other content types are possible too.

In addition to `GET` and `POST`, HTTP supports several other request methods, but they are far less widely used. They are:

- `HEAD` – identical to a `GET`, but omit the response body, so that we just receive meta-data about the resource such as its size and last-modified time **(useful)**
- `PUT` – create or replace a named resource on the server **(sometimes useful)**
- `DELETE` – remove a named resource from the server **(sometimes useful)**
- `PATCH` – make partial updates to a resource on the server **(rarely useful)**
- `CONNECT` – has something to do with switching to a secure protocol, but not often used **(ignore)**
- `OPTIONS` – determine what methods the server supports **(ignore)**
- `TRACE` – has something to do with proxy servers **(ignore)**

Different subsets of these methods are defined to be **safe** and **idempotent**.

- **Safe** means that the method should not have any significant consequence other than retrieval of information. Safe methods include `GET`, `HEAD`, `OPTIONS`, and `TRACE`.
- **Idempotent** means multiple identical requests should have the same effect as a single request. All the safe methods are also idempotent, but also `PUT` and `DELETE`.

Note that `POST` is explicitly **not** idempotent, which is why we reserve it for actions that absolutely should not be repeated, such as charging customers' credit cards.

Responses

After the server has received and processed the client's request, it will issue its own response. Responses begin with a single **status line** that contains three parts:

- The protocol **version,** such as `HTTP/1.0`
- The numeric **status code,** a three-digit number such as `200` or `404`
- The human-friendly **reason phrase,** a short message such as `Not Found` or `Internal Server Error`

Following the status line is a **headers** section. Commonly used headers include:

- `Content-Type:` The type of data to be transmitted in the payload. Often `text/html` for an HTML page but can be `text/plain` or other data formats like `image/jpeg`
- `Expires:` Provides a date until which this content can be cached (stored and reused without issuing a new request).

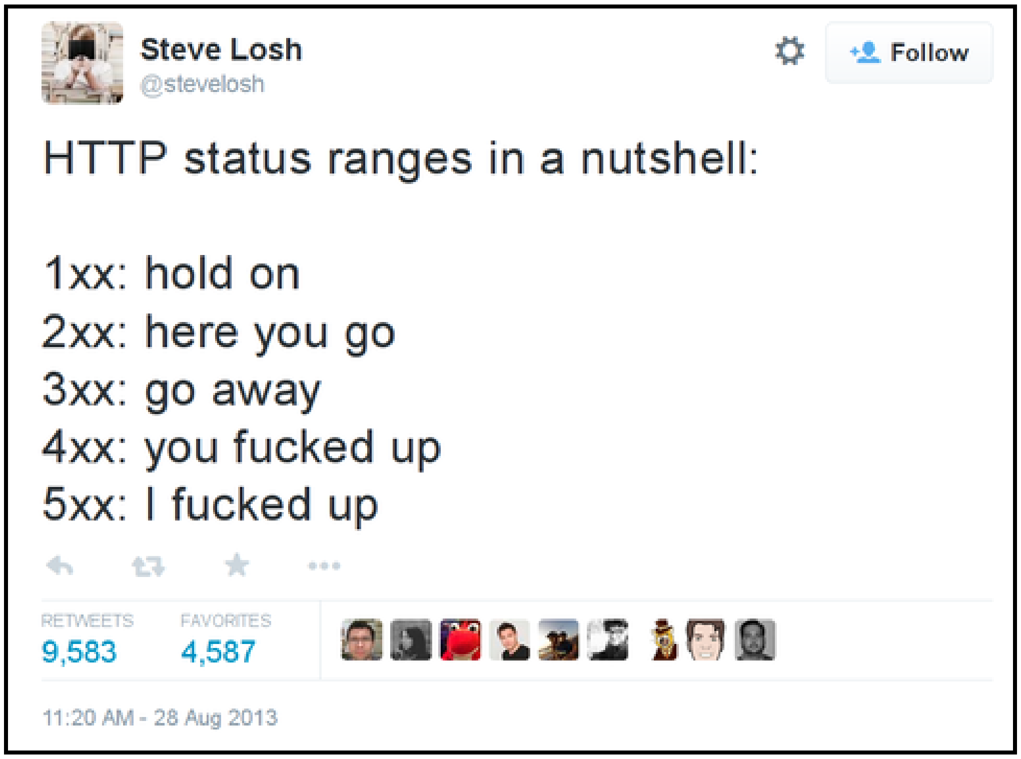
- `Server:` A string that identifies the server software and version – serves the same purpose as the `User-Agent` header that identifies the client software.
- `Set-Cookie:` Provides some data to be returned to the server on the next request, to help implement **sessions** – we'll learn more about that soon.

Here is a status line and complete set of headers for a request that my Firefox browser made to `metamodpro.com` for that language-codes page:

```
HTTP/1.1 200 OK
Cache-Control: post-check=0, pre-check=0
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Mon, 26 Jan 2015 22:06:27 GMT
Expires: Mon, 1 Jan 2001 00:00:00 GMT
Keep-Alive: timeout=5, max=100
Last-Modified: Mon, 26 Jan 2015 22:06:27 GMT
P3P: CP="NOI ADM DEV PSAi COM NAV OUR OTRo STP IND DEM"
Pragma: no-cache
Server: Apache
Transfer-Encoding: chunked
X-Content-Encoded-By: Joomla! 1.5
X-Powered-By: PHP/5.3.29
```

The numeric response codes are organized into several categories, identified by the first digit. I'll show only the most well-known commonly-used codes here; the complete list is available from Wikipedia or the W3.

- Codes beginning with `2` (indicated as `2xx`) declare a successful transaction:
  - `200 OK` is the most common, by far. You can see that in the extended example above.
  - `201 Created` is used just to indicate the creation of a new resource on the server (used with the `PUT` method).
- Codes such as `3xx` indicate some form of **redirection** – for example, they ask the client to repeat the request at a different URL.
  - `302 Found` provides a new URL in the `Location` header.
  - `304 Not Modified` is used when the client's request contained an `If-Modified-Since` header, and the resource has not been modified, so the client should continue to use its cached copy.
- The `4xx` indicate an error on the part of the client.
  - `400 Bad Request` means that somehow the syntax of he request was not understood.
  - `403 Forbidden` means that access has been denied to the requested resource.



Steve Losh
@stevelosh

HTTP status ranges in a nutshell:

1xx: hold on
2xx: here you go
3xx: go away
4xx: you fucked up
5xx: I fucked up

RETWEETS 9,583 FAVORITES 4,587

11:20 AM - 28 Aug 2013

Figure 2: @stevelosh on Twitter

  - `404 Not Found` means the requested resource could not be found, but may be available again in the future.

- Finally, `5xx` indicate an error on the part of the server.
  - `500 Internal Server Error` is a completely generic error message. If the server-side program crashes or experiences a run-time error, this is the typical response.
  - `501 Not Implemented` means the server lacks the ability to fulfill the request, but it may be implemented in the future.
  - `503 Service Unavailable` means the server is currently unavailable, because it is overloaded or down for maintenance.

The curl command

The `curl` command is an indispensable tool for web developers. It allows you to issue highly-customized HTTP requests directly from the command line, bypassing the web browser.

- On Windows, it was installed along with Git, so you run it by opening the **Git bash** application.
- On Mac, it is included with the OS, so you run it in the **Utilities** » **Terminal** appli-

cation.

Either way, when you enter `curl` at the terminal prompt, you should get this informational message:

```
curl: try 'curl --help' or 'curl --manual' for more information
```

The simplest use of `curl` is just to provide a URL on the command line. It will issue a `GET` request, and then dump the payload (**response body**) into your terminal. For example, try:

```
curl http://www.google.com/
```

You'll get a whole mess of HTML and Javascript code, probably ending with `</script></div></body></html>`.

By specifying the `-I` option (that's the capital letter that rhymes with 'eye'), you instruct `curl` to do a `HEAD` instead of `GET` and show the response headers instead of the payload.

```
curl -I http://www.google.com/
```

On my system, the result was:

```
HTTP/1.1 200 OK
Date: Mon, 26 Jan 2015 22:52:58 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=4ae6a2df633f4c61:FF=0:TM=1422312778:LM=1422312778:S=WkV9
    8-7Qk9-y01NS;expires=Wed, 25-Jan-2017 22:52:58 GMT; path=/; domain=.goog
    le.com
Set-Cookie: NID=67=TCZT1yid-KNBAX4NqXJ8QVKIp48mGjzmBFYYE_d9rdvybLTQqNgsID13Y
    mCssBG54kRC7kLAVeLokFrOBNmzh4-kfVX5C4LPXzi2DBFYvZUArv3yJS0aSqaE_uNSsPV0;
    expires=Tue, 28-Jul-2015 22:52:58 GMT; path=/; domain=.google.com; HttpO
    nly
P3P: CP="This is not a P3P policy! See http://www.google.com/support/account
    s/bin/answer.py?hl=en&answer=151657 for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic,p=0.02
Transfer-Encoding: chunked
Accept-Ranges: none
Vary: Accept-Encoding
```

Adding the `-v` option will show the (mostly) complete conversation between client and server. For example, let's try it, but also remove the `www.` from the URL, so we try access `google.com` directly.

```
curl -v http://google.com
```

The result is large, but manageable. Lines starting with `*` are debugging messages from `curl` itself. Then data sent by the client is marked > and by the server is <. Finally, between `<HTML>` and `</HTML>` is the response body.

```
* Rebuilt URL to: http://google.com/
*   Trying 173.194.123.37...
* Connected to google.com (173.194.123.37) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.40.0
> Host: google.com
> Accept: */*
>
< HTTP/1.1 301 Moved Permanently
< Location: http://www.google.com/
< Content-Type: text/html; charset=UTF-8
< Date: Mon, 26 Jan 2015 22:59:11 GMT
< Expires: Wed, 25 Feb 2015 22:59:11 GMT
< Cache-Control: public, max-age=2592000
< Server: gws
< Content-Length: 219
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< Alternate-Protocol: 80:quic,p=0.02
<
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
* Connection #0 to host google.com left intact
```

You can see that this request resulted in a `301 Moved Permanently` response, and the `Location` header tells us we should access `http://www.google.com/` instead. (You can add the `-L` option to ask `curl` to follow these `3xx` redirect messages automatically.)

Another fancy trick with `curl` is we can force it to send particular **headers** in its request, using the `-H` option. **You must put the header content in quotes.** For example, if you want to use the `Accept-Language` header to get a Spanish version of the Google web page:

```
curl -H "Accept-Language: es" http://www.google.com/
```

It still looks like a huge pile of HTML and Javascript, but if you trawl carefully you can find phrases like *Iniciar sesión* (log in), *Noticias, Imágenes* (for news and image search), and an ad for Google Chrome: *Una forma más rápida de navegar la web.* (Although sometimes the accented characters won't appear properly in the terminal, depending on your configuration.)

We'll cover one more trick that `curl` can do: specifying form parameters for a `POST` request, using `-d` and then a quoted variable assignment. This is part of how to do check-ins 1 and 2, so we'll use the URL provided there as an example:

```
curl -v -d "name=My+Name" http://cs120.liucs.net/checkin
```

With the `-v` we get to see (almost) the entire conversation, but I'll abbreviate it slightly here:

```
> POST /checkin HTTP/1.1
> User-Agent: curl/7.40.0
> Host: cs120.liucs.net
> Accept: */*
> Content-Length: 12
> Content-Type: application/x-www-form-urlencoded
>
< HTTP/1.1 400 Bad Request
< Server: Warp/3.0.5
< Content-Type: text/plain; charset=utf-8
<
InvalidArgs ["Missing required parameter: score"]
```

The request went through to the server, and it sent 12 bytes of payload (count up the number of characters in `name=My+Name`). The response was `400 Bad Request` because you are also expected to specify a parameter `score`. See the Check-in 2 spec for more details.

To specify additional parameters, use a separate `-d` for each.

Further resources

- Video: WTF is HTTP [7:42] especially covers the difference between GET and POST methods.
- A limerick by classam on Twitter – see also HTCPCP

    I wanted a beverage, hot
    From an HTTP coffeebot.

> My coffee was spurned
> An error returned:
> 418 I AM A TEAPOT

- The clever 404 and 500 pages at Bloomberg