Chapter 1: HTTP

HyperText Transport Protocol, this protocol beat FTP and Gopher because it was more flexible, extensible, and not burden with licensing agreements. It sends content not files which is useful if you are just searching for things and don't want to download an entire file or a small subset of that file. Can respond to requests based on reserved VERB words like GET, POST, DELETE, PUT, etc The flexibility comes from the fact custom headers could be included allowing for extension and adding new features. The protocol allowed for a more request or command oriented pattern for communication. HTTP relied on the pairing of web clients and web servers as well as using URIs to describe resources, allowing more than 1 resources to be hosted on 1 server. The standard for HTTP is outlined by RFC2616.

Generally HTTP uses TCP, and on port 80. For HTTPS to allow encrypted HTTP traffic using TLS it uses port 443 (generally). HTTP can work on both IPV4 and 6 addresses. HTTP requests are made to addresses called **URIs**.

HTTP Command

Every HTTP command is made to a URI

Command	What it does
GET	Retrieve information from that URI
POST	Run search, log-in, append data, change data
HEAD	GET without a message body (for caching)
PUT	Store the entity at that URI
DELETE	Delete the resource at that URI
PATCH	Modify the entity at that URI
OPTIONS	What options a resource can accommodate
TRACE	Debugging or echo request
CONNECT	Tunnelling proxy over HTTP

PATCH is not defined in RFC2616 those are in a different RFC standard.

URI

A URI is an Universal Resource Identifier, it identities i.e., points to a resource. Most URIs are URLs or Uniform Resource Locator but not all URLs are URIs. URLs tells you how to get to a resource. Some URIs are URNs or Uniform Resource Name, they tell you the unique name or number given to a resource by some body like IETF

URL

As stated before URLs or Uniform Resource Locator tells you how to get to a resource. It is comprised with two parts:

- **Scheme** e.x., - http, https, mailto, file, data, ftp, gopher, irc, spotify
- **Everything else** - Anything not part of the scheme.

URL syntax

All URLs follow a fixed syntax

```
URI =
scheme:[//authority]path[?query][#fragment]
```

The authority follows another syntax listed below

```
authority = [userinfo@]host[:port]
```

Syntax	Example
scheme	http, https
authority	Can be a username and password as part of userinfo, Host followed by a port after :
path	The location of the resource
?	A query string to query the resource
#	Fragments that will provide directions to a secondary resource such as a section heading within the resource

Absolute and Relative URLs

Both the authority and path can be absolute or relative, if the authority is missing then the authority is implied based on the current URL. If the path has any `..` or is relative then it locates the resource based on the "current working director" of the current URL. Examples:

- Absolute authority, absolute path - http://[::1]:8000/images/web-server.svg
- Absolute authority, relative path - http://[::1]:8000/images/../index.html
- Implied authority, absolute path - /images/web-server.svg
- Implied authority, relative path - image/web-server.svg

Queries

Queries are an optional portion that can be added to a URL. You can have one or more arguments in a key-value pair format like so `key=value&key2=value2`. Other separator exist like `;` instead of `&` or just having a string and no key-value pairs. The syntax is quite loose and it's highly dependent on the webserver.

Fragments

Fragments are an optional portion that can be added to a URL. They allow jumps to some spot in the content like a time in a video using `#t=` in a YouTube link, or a part of a page like a section's name.

Encodings

Universal URIs have to be able to handle anything, including characters like spaces, accents, punctuations, emoji, etc. For HTTP assume our URLs are Unicode UTF-8 encoded. For characters that aren't in `-._ 0-9a-zA-Z` we use `%` encoding. Uses RFC 3986, space is encoded as `%20`. For domina names we use "punycode" encoding where emojis and other special characters are converted to visible ASCII.

HTTP 2

Released in 2015, it is based on the SPDY protocol by Google and it was designed to reduce latency. It only supported over TLS so only on HTTPS connections and is backwards compatible with HTTP version 1's methods, including their status codes, headers, and URIs. Instead of encoding everything as ASCII, the entire protocol communicates in binary to reduce bandwidth. Features that make it faster, and more responsive (lower latency) is:

- **Pipelining** - Allows multiples requests to be handled at the same time instead of waiting for a response before continuing.
- **Push** - Allow the servers to look into the future and sends clients content before they even request it
- **Multiplexing** - Allows different types of requests to be send back to the client at the same time via interleaving.

This avoids using multiple TCP connection to a single server thus reducing the number of high-latency TCP and TLS handshakes. Both request and response is happening at the same time and data is interleaved. Requests and responses can be prioritized.

HTTP Request

To send a message to a **server** *from* the **client** one would need to send a HTTP Request message.

Definition 1.0.1 (HTTP Request). *This is the grammar for an HTTP request.*

```
Request = Request-Line
          *(( general-header
           | request-header
           | entity-header )"\r\n")
          "\r\n"
          [ message-body ]
```

Where:

```
Request-Line = Method URI HTTP-Version"\r\n"
```

The request line always come first where the `Method` is the type of request, the `URI` is the "URL" and the `HTTP-Version` is what version of HTTP you are running. For format for `HTTP-Version` is `HTTP/X.Y` where `X` is the major and `Y` is the minor version numbers. For this project it's entire in HTTP1.1 so `HTTP-Version` should be `HTTP/1.1`.

There must be an empty line with a `CRLF` or `\r\n` right after the header to separate the header and the body of the message.

HTTP Response

To receive a message from the server or responding to a request from the server-side one would need to send a HTTP Response message.

Definition 1.0.2 (HTTP Response). *This is the grammar for an HTTP response.*

```
Response = Status-Line
           *(( general-header
            | response-header
            | entity-header )"\r\n")
           "\r\n"
           [ message-body ]
```

Where:

```
Status-Line = HTTP-Version Status-Code
Reason-Phrase"\r\n"
```

Like in HTTP Request the status line comes first where the `Status-Code` is the response code and the `Reason-Phrase` is the phrase associated with the `Status-Code`. The format is for the HTTP-Version is the same as in HTTP Request.

HTTP Headers

There are a lot of different headers that can make up a HTTP message but for all headers they follow a specific syntax

```
header = field-name":" [ field-value ]
```

It's encoded like a key value pair where `field-name` is the header's name and the `field-value` is the value for that header.

GET

HTTP GET is a simple request to be sent that resource.

- It might be dynamic (resource is generated when the request is received by the software on the server), this could be done by a JavaScript or a Python CGI.
- It might be static (just a file sitting on the server)
- It might be a mix of static and dynamic content.

We can send query parameters along with a HTTP GET in the URL using the `?` character. GET request **SHOULD NOT** cause the server to change data-we should use a different HTTP method for that. GET request are only used for reading data/information.

Note

You **SHOULD NOT** send a body with a GET Request because it might lead to a rejection. The behaviour is undefined so to keep everything working smoothly for all platforms you shouldn't send one.

Request Headers

With HTTP 1.1 you **must** send a `Host` header with the authority you are connecting with as the value. You **should** also send these headers as well

- Connection: Keeps the connection open or not after the current transaction finishes.
- Date: The date of the request.
- Accept: Advertises which content type, expressed as MIME types the client is able to understand.

POST

HTTP POST is a request to update, create, or generally interact with a URL.

- Can do things like queries like HTTP GET but not limited by length.
- Use to submit HTML forms.
- POST is expected to add or mutate data.

Moreover POST:

- URI identities a service/handler/script/process
- Arguments are stored in the HTTP request body
- The request body is interpreted by some software and processed
- "Send this here for processing" basically

POST can be used for:

- Login/logout
- Reply
- Post on a forum/blog
- Upload multiple files (Somewhere)
- Make an order
- Fill out a survey/poll

Forms

POST methods often send their parameters in the body of the POST requests. The `Content-Type` of the body is usually some type of form

- `applicaiton/x-www-form-urlencoded`
- `multipart/form-data` - Generally used when uploading files

Each form element has a `name`. The `name` is the identifier used for the variable that is going to store the content. Example: `<input name="occupation">` becomes `occupation=` The content of the input element becomes the value.

```
POST /test HTTP/1.1
Host: foo.example
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

field1=value1&field2=value2
```

Figure 1.1: Example `x-www-form-urlencoded` https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

multipart/form-data uses mime (multipurpose internet mail extensions) to send form data. Mostly used to upload files as binary, but it can be used for any form. Send the content-size first and then ask the server i that's okay. The server must response with a `100 Continue` code if it can handle that size of data. Then the client will send the body. This interaction is a bit slower because it adds a round trip latency for confirmation with the header before sending the body. But the confirmation is done to ensure that the client doesn't send a file that is too big for the server to handle or of a type it can't handle, etc.

```
POST /test HTTP/1.1
Host: foo.example
Content-Type: multipart/form-data;boundary="boundary"

--boundary
Content-Disposition: form-data; name="field1"

value1
--boundary
Content-Disposition: form-data; name="field2"; filename="example.txt"

value2
--boundary--
```

Figure 1.2: Example `mutlipart/form-data` https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

The boundary lets the server tell when one part of the form ends and another begins. The random number should be chosen so that it won't show up in the content. The last boundary line **must** end with `--` to tell the server that all of the form has been sent.

Request Headers

With HTTP 1.1 you **must** send these headers

- Host: Specifies the host and port to which the request is being sent to.
- Content-Type: The type of the form or what type of the body.
- Content-Length: The size of the body.

You **should** also send these headers as well

- Connection: Keeps the connection open or not after the current transaction finishes.
- Date: The date of the request.
- Accept: Advertises which content type, expressed as MIME types the client is able to understand.