

Jonathan Klinger

5BHIF

Einfacher HTTP 1.1 Client

A short introduction to HTTP 1.1, it's requests
and a simple Client written in C++

Abteilung Informatik
HTBLuVA Wiener Neustadt
Jänner 2021

Contents

1	HTTP	2
1.1	Eigenschaften	2
1.1.1	(Relativ) Einfach	2
1.1.2	Zustandslos	2
1.1.3	Nicht auf Hypertext beschränkt	2
1.1.4	Verhandlung der Datenr�presentation	2
1.2	HTTP 1.1	2
1.3	Funktionsweise	3
1.3.1	Header	3
1.4	HTTP Authentifizierung	4
1.5	HTTP Status Codes	4
1.6	HTTP Anfragen	4
1.6.1	GET	4
1.6.2	DELETE	4
1.6.3	POST	5
1.6.4	PUT	5
2	Das Programm	5
2.1	Aufgabenstellung	5
2.2	Meine L�sung	5
2.2.1	Asio	5
2.2.2	Spdlog	5
2.2.3	CLI11	6
2.2.4	JSON	6
2.2.5	Ablauf	6
2.2.6	Schwierigkeiten	11
2.2.7	Bedienung	12
3	Sources	12

1 HTTP

HTTP steht für Hyper Text Transfer Protocol. Es handelt sich um ein Client/Server Protocol das ursprünglich zum Abrufen von statischen Informationen (z.B. Browser ruft eine Website ab) gedacht war. Das Protokoll ist in der Anwendungsschicht (Application Layer) des OSI-Modells angesiedelt. Außerdem wird auf darunterliegenden Schichten mit TCP gearbeitet, was es zuverlässig macht.

HTTP wurde von der Internet Engineering Task Force (IETF) und dem World Wide Web Consortium standardisiert.

1.1 Eigenschaften

1.1.1 (Relativ) Einfach

Das Protokoll arbeitet mit einfachen Anfragen und Antworten (Requests und Responses). D.h. der Client (z.B. wieder ein Webbrowser) schickt eine Anfrage, der Informationen über die Ressource die er zugeschickt bekommen möchte an den Server, der mit einer standardisierten Antwort, die im besten Fall die Ressource enthält, antwortet. Auf die verschiedenen Arten von Request und die Arten von Response Status Codes werde ich später eingehen.

1.1.2 Zustandslos

HTTP ist Zustandslos, was bedeutet, dass alle Anfragen unabhängig voneinander sind. Informationen aus vorherigen Anfragen gehen verloren. Mittels Cookies in den Header-Informationen (siehe unten) können allerdings Status-Informationen zugeordnet werden.

1.1.3 Nicht auf Hypertext beschränkt

Auch wenn es ursprünglich dafür gedacht war, können immer mehr beliebige Daten versendet werden.

1.1.4 Verhandlung der Datenrepräsentation

Die Content Negotiation wird verwendet um eine Abstimmung der Inhalte einer Ressource auf die Möglichkeiten des Clients ermöglicht.

1.2 HTTP 1.1

Diese Version des Protokolls wurde 1999 im RFC2616 publiziert. Eine der größten Erweiterungen war, dass der Client den Wunsch äußern kann (per Header Eintrag, siehe unten) den Verbindungsabbau zu überspringen, um dieselbe später wieder nutzen zu können. Seitdem kann auch eine TCP Verbindung für mehrere Anfragen verwendet werden können was die Ladezeiten extrem verkürzt da diese gerade beim Verbindungsaufbau langsam sind.

Außerdem wurde der MIME-Typ multipart/replace hinzugefügt.

1.3 Funktionsweise

Wie oben schon angeschnitten, funktioniert HTTP mittel Anfrage und Antwort. Egal ob man im Browser eine URL eingibt oder ob man selbst mittels verschiedenen Technologien Anfragen versendet läuft der Ablauf zwischen Client und Server gleich ab.



1.3.1 Header

Die notwendigen Informationen für eine erfolgreiche Kommunikation werden im Header hinterlegt.

Anfrage Header:

```
GET /maps HTTP/1.1
HOST: www.google.com
Connection: close
...
```

Notwendig sind nur die 1. Beiden Zeilen. "GET" spezifiziert den Anfrage-Typ (siehe unten) danach folgt der Pfad am Server zur Ressource und die verwendete HTTP-Version. Die nächste Zeile spezifiziert den Host (und Portnummer) mittels URL.

Antwort Header:

```
HTTP/1.1 200 OK
Content-Type: text/html
Date: Fri, 16 Apr 2021 00:00:00 GMT
Server: Apache
Set-Cookie: key=value
...
```

Der Header der Antwort ist meistens um einiges umfangreicher als der der Anfrage. In der Ersten Zeile ist wieder die verwendete HTTP Version sowie der Status Code und die Status Message. Es folgen ein Auszug der wichtigsten Antwort-Header-Felder.

Der Content-Type spezifiziert wie die Daten die nach dem Header folgen interpretiert werden sollten. Date enthält das Datum und die genaue Zeit wann die Antwort gesendet wurde. Das Feld Server zeigt an welche Software der Server verwendet. Und mit Set-Cookie werden die bereits erwähnten Cookies gesetzt.

1.4 HTTP Authentifizierung

Grundsätzlich gibt es mehrere Arten der Authentifizierung bei diesem Protokoll. Hier werde ich nur auf die Basic Authentication eingehen. Ist eine Ressource geschützt fordert der Server eine Benutzername/Passwort-Kombination an. Wird diese nicht geliefert sendet der Server einen Fehlercode (401, siehe unten). Die Kombination wird im Header folgendermaßen angegeben:

```
Authentication: Basic YmVudXR6ZXI6cGFzc3dvbnQ=
```

Der hier zu sehende Text ist **nicht** verschlüsselt. Es handelt sich lediglich um die Kombination in Base64 codiert. Die Kombination muss als "benutzername:passwort" codiert werden.

Da es sich wie erwähnt nur um Base64 handelt ist diese Methode sehr unsicher!

1.5 HTTP Status Codes

Jede Anfrage wird vom Server mit einem passenden Statuscode und einer Message beantwortet. Es kann z.B. über den Erfolg oder etwaige Fehler informiert werden.

Hier ein Überblick über die möglichen Antworten:

- 1xx - Info z.B.: 100: Continue
- 2xx - Erfolg z.B.: 200: OK
- 3xx - Umleitung z.B.: 300: Multiple Choice
- 4xx - Client Fehler z.B.: 401: Unauthorized
- 5xx - Server Fehler z.B.: 500: Internal Server Error

Mithilfe der Fehlercodes kann der Client auf verschiedene Antworten passend reagieren und gegebenenfalls eine Fehlersuche erleichtern.

1.6 HTTP Anfragen

1.6.1 GET

Ist die am meisten verwendete Methode. Es wird eine Ressource vom Server angefordert. Es besteht die Möglichkeit Argumente im URI zu übermitteln. Laut Standard sollte mit GET nur Ressourcen abgerufen werden, sonst nichts. Der Request enthält keinen Body und somit auch keine Header Elemente wie Content-size.

1.6.2 DELETE

Löscht die angegebene Ressource auf dem Server. Der Request enthält ebenfalls keinen Body und somit auch keine Header Elemente wie Content-size.

1.6.3 POST

Schickt unbegrenzte Mengen an Daten zur weiteren Verarbeitung an den Server. Diese Daten befinden sich im Body der Nachricht. Dieser Inhalt kann einfacher Text, Key-Value-Paare oder ähnliches sein. Die Daten werden üblicherweise nicht gecached.

1.6.4 PUT

Liefert die gleiche Funktionalität wie POST mit dem Unterschied, dass wenn bereits eine Ressource bereits existiert ersetzt und sonst neu erstellt wird. Der Vorgang ist somit idempotent.

2 Das Programm

2.1 Aufgabenstellung

Die Aufgabenstellung:

“Funktionsfähiger HTTP 1.1-Client zum Herunterladen/Speichern von 3 beliebigen Dateien (GET, PUT, POST, DELETE, Basisfunktionalität; Steuerung über Kommandozeile; inkl. HTTP Basic-Authentication, nicht auf Header vergessen und inkl. Cookies, http-parser kann verwendet werden) “

2.2 Meine Lösung

Ich habe meine Arbeit in C++ programmiert. Es gibt einige Lösungsansätze. Für Übungszwecke habe ich mich auf die Verwendung von asio und spdlog konzentriert.

2.2.1 Asio

Asio ist eine Plattform-übergreifende Bibliothek für Netzwerkprogrammierung. Die Bibliothek stellt eine Reihe an synchronen und asynchronen Funktionen zur Verfügung.

Synchrone Kommunikation basiert auf blockierenden Operationen, während Asynchrone auf nicht blockierenden Operationen und Callbacks basiert.

Asio bietet die Möglichkeit mit Fehlercodes oder auch Exception die Fehler handzuhaben.

2.2.2 Spdlog

Bei spdlog handelt es sich um eine header only Bibliothek die das Logging erleichtert. Die Vorteile liegen darin das es sehr schnell arbeitet und eine schöne, farbige Formatierung ermöglicht, die auch individuell angepasst werden kann.

2.2.3 CLI11

Hier handelt es sich um einen Kommandozeilen-parser. Es ist wie auch spdlog eine schnelle Header-only-Bibliothek. Ein großer Vorteil ist das diese Bibliothek plattformübergreifend funktioniert. Es lassen sich auch schöne Fehlermeldungen bilden.

2.2.4 JSON

Die JavaScript Object Notation ist ein Datenformat in Textform und dient zum Datenaustausch zwischen Anwendungen. Die gleichnamige Header-Only Bibliothek von dem Github-User nlohmann (siehe <https://github.com/nlohmann/json>) liefert die Funktionalität um diese Datenart zu lesen, interpretieren und weiter zu verarbeiten.

In diesem Projekt wird mit der Hilfe dieser Bibliothek aus einer JSON-Datei voreingestellte Benutzernamen und Passwörter für die jeweiligen Anfragen festgelegt.

2.2.5 Ablauf

Das Programm kann 3 Anfragen pro Aufruf durchführen. Diese werden per CLI11 aufgenommen. Jeder wert von jeder einzelnen Anfrage wird als Option im Aufruf des Programms aufgeführt. Diese Optionen werden in je in einer Variable gespeichert. Die Variablen jeder Anfrage werden wieder in einem Vector zusammengefasst der dann der Funktion die die eigentliche Anfrage sendet übergeben werden.

```
1//Interface
2CLI::App app{"Einfacher HTTP1.1 Client"};
3
4//Erster Request
5string type1 = "";
6string url1 = "";
7string port1 = "80";
8string path1 = "/";
9string file1 = "test.txt";
10string user1 = "";
11string pw1 = "";
12string cookies1 = "";
13string content_type1 = "";
14string content1 = "";
15
16//Standardauthentifizierung aus json
17user1 = j["user1"];
18pw1 = j["pw1"];
19
20app.add_option("--type1", type1, "Typ von Request 1");
21app.add_option("--url1", url1, "URL von Request 1");
22app.add_option("--port1", port1, "Port von Request 1");
23app.add_option("--path1", path1, "Path von Request 1");
24app.add_option("--file1", file1, "Dateiname von Request 1");
25app.add_option("--user1", user1, "User von Request 1");
26app.add_option("--pw1", pw1, "Password von Request 1");
```

```

27app.add_option("--cookies1", cookies1, "Cookies von Request 1");
28app.add_option("--contentType1", content_type1, "Content Type von
    Request 1");
29app.add_option("--content1", content1, "Content von Request 1");
30
31CLI11_PARSE(app, argc, argv);
32
33vector<string> req1{type1, url1, port1, path1, cookies1,
    content_type1, content1, file1, user1, pw1};

```

In diesem Ausschnitt sieht man die Initialisierung der Variablen für die Erste Anfrage, sowie die Zusammenfassung in einem Vector. Die Reihenfolge muss in diesem Fall *nicht* stimmen (siehe unten). mittels "app.add_option()" wird die Benutzereingabe den jeweiligen Variablen zugewiesen. In den Zeilen 17 und 18 werden die Standard- Benutzer und Passwörter zugewiesen. Falls der Benutzer jedoch diese im Aufruf angeben möchte werden diese einfach ersetzt.

Das Auslesen der JSON datei funktioniert folgendermaßen:

```

1json j;
2ifstream config_file("config.json");
3string line, config;
4while (getline (config_file, line)) {
5    config += line;
6}
7
8config_file.close();
9
10try {
11    j = json::parse(config);
12} catch (json::parse_error& ex) {
13    std::cerr << "parse error at byte " << ex.byte << std::endl;
14}

```

Mithilfe der C++ Standardbibliothek fstream wird der Inhalt der Datei ausgelesen. Die Funktion json::parse() übersetzt den Inhalt dann in das JSON-Format und macht es für den Programmierer zugänglich.

Diese Benutzereingabe wird jetzt so gut als Möglich auf Fehler überprüft. In der main-Funktion wird sie hinsichtlich des Anfragetyps getestet.

```

1if (type1 == "GET" || type1 == "DELETE") {
2    send_GET_DELETE(req1, "1");
3} else if (type1 == "POST" || type1 == "PUT") {
4    send_POST_PUT(req1, "1");
5} else {
6    spdlog::error("1. Request: Typ nicht erkannt");
7}
8
9if (type2 != "") {
10    if (req2[0] == "GET" || req2[0] == "DELETE") {
11        send_GET_DELETE(req2, "2");
12    } else if (req2[0] == "POST" || req2[0] == "PUT") {
13        send_POST_PUT(req2, "2");
14    } else {
15        spdlog::error("2. Request: Typ nicht erkannt");
16    }
}

```



```

17 }
18
19 if (type3 != "") {
20     if (req3[0] == "GET" || req3[0] == "DELETE") {
21         send_GET_DELETE(req1, "3");
22     } else if (req3[0] == "POST" || req3[0] == "PUT") {
23         send_POST_PUT(req3, "3");
24     } else {
25         spdlog::error("3. Request: Typ nicht erkannt");
26     }
27 }

```

Es wird davon ausgegangen das zumindest eine Anfrage durchgeführt wird. Die anderen Vektoren werden geprüft ob sie gesetzt sind, da sonst der Fehler "segmentation fault (core dumped)" aufgerufen wird da versucht wird mit nicht gestztem Speicher zu arbeiten. Wird kein passender Anfragetyp erkannt, wird mit spdlog eine Fehlermeldung angezeigt. Auch wenn andere Anfragen fehlerhaft sind, werden trotzdem die anderen durchgeführt.

Die Restliche Eingabe wird in den Funktionen "send_GET_DELETE()" und "send_POS_PUT()" analysiert und verarbeitet.

Die variable

```
1 unsigned int file_place = 4;
```

zeigt die Position an der der Dateiname steht in dem die Antwort auf eine GET Anfrage gespeichert werden soll. Bei POST erübrigt sich das, da nicht in Dateien gespeichert wird. Diese Variable ist allerdings nicht mehr notwendig, da der Dateiname jetzt einen fixen Platz hat.

Ursprünglich hat dieser if/else Teil anhand der länge der Eingabe analysiert welche weiteren Header Felder gestzt und die variable file_place angepasst werden muss.

```

1 if (input.size() == 8) {
2     spdlog::info(n + ". Request: Cookies erkannt");
3     spdlog::info(n + ". Request: HTTP Basic Authorization erkannt");
4     ;
5     file_place = 7;
6
7     string auth = input[5] + ":" + input[6];
8     string base_auth = base64(auth);
9
10    req_string = req_string +
11    "Authorization: Basic " + base_auth + "\r\n"
12    "Cookie: " + input[4] + "\r\n\r\n";
13} else if (input.size() == 7) {
14    spdlog::info(n + ". Request: HTTP Basic Authorization erkannt");
15    ;
16    file_place = 6;
17
18    string auth = input[4] + ":" + input[5];
19    string base_auth = base64(auth);

```

```

20
21     req_string = req_string + "Authorization: Basic " + base_auth +
        "\r\n\r\n";
22} else if (input.size() == 6) {
23     spdlog::info(n + ". Request: Cookie erkannt");
24     file_place = 5;
25
26     req_string = req_string + "Cookie: " + input[4] + "\r\n\r\n";
27} else {
28     req_string = req_string + "\r\n";
29}

```

Da diese Methode auf der alten Version der Benutzereingabe die auf Positionen basierte ruht, wurde mit dem Überarbeiten der Benutzereingabe auch dieser Teil aktualisiert:

```

1 if (input[4] != "") {
2     spdlog::info(n + ". Request: Cookies erkannt");
3
4     req_string = req_string + "Cookie: " + input[4] + "\r\n";
5 }
6 if (input[8] != "" && input[9] != "") {
7     spdlog::info(n + ". Request: HTTP Basic Authorization erkannt");
8
9     string auth = input[8] + ":" + input[9];
10    string base_auth = base64(auth);
11
12    req_string = req_string + "Authorization: Basic " + base_auth + "
        \r\n";
13} else if ((input[8] == "" && input[9] != "") || (input[8] != "" &&
        input[9] == "")) {
14    spdlog::error(n + ". Request: Username/Passwort nicht erkannt");
15}
16
17req_string = req_string + "\r\n";

```

Das ist ein Auszug aus der Funktion `send_GET_DELETE()`. In der verbesserten Version werden nur die einzelnen Optionen auf ihren Inhalt geprüft und dementsprechend der HTTP-Header erweitert. Wie man erkennen kann ist der Code jetzt lesbarer und kürzer.

Die Hier gezeigte Funktion `"base64()"` kümmert sich um die oben beschriebene codierung von `benutzer:passwort`. Da es keine C++ Funktion dafür gibt und ich in keiner der zur Verfügung stehenden Bibliotheken eine passende funktion gefunden habe, habe ich auf die Linux eigene bash Funktion mithilfe eines System-calls zugegriffen. Das funktioniert folgendermaßen:

```

1 string base64(string str) {
2     string command = "printf " + str + " | base64 > base64.txt";
3     system(command.c_str());
4
5     ifstream ifs("base64.txt");
6     string ret{ std::istreambuf_iterator<char>(ifs), std::
        istreambuf_iterator<char>() };
7     ifs.close();
8     return ret;

```

```
9}
```

Mithilfe des Systemcalls wird der codierte text in eine Datei geschrieben die ich dann auslesen kann. Leider Funktioniert das Programm damit nur auf Linux-Betriebssystemen, was aber in diesem Fall kein Problem ist.

Die Anfrage wird mithilfe von Asio folgendermaßen gesendet. Hierbei handelt es sich um die vorherige Version.

```
1char reply[1000];
2error_code ec;
3
4size_t reply_length = asio::read(sock, asio::buffer(reply), ec);
5
6//Char[] to String
7string res = "";
8for (unsigned int i=0; i<reply_length; i++) {
9    res = res + reply[i];
10}
11//Only get body/status from response
12string body = res.substr(res.find("\r\n\r\n") + 4);
13string status = res.substr(0, res.find("\r\n"));
```

In dem character Array reply wird die Antwort des Servers gespeichert. Diese wird dann wieder in einen string umgewandelt damit man die funktion "find()" verwenden kann.

In der verbesserten Version wird am senden an sich nichts geändert aber Fehler werden abgefangen. Das sieht so aus:

```
1size_t reply_length = asio::read(sock, asio::buffer(reply), ec);
2if (ec.value() != 2) {
3    spdlog::error(n + ". Request: fehlgeschlagen!");
4}
```

Der Value 2 steht in diesem Fall für den Erfolg der Anfrage. Alles was nicht Erfolg ist wird hiermit abgefangen und der Benutzer wird darüber mit einer Fehlermeldung informiert.

Mit den Variablen "status" und "body" wird dann eine Ausgabe mittels spdlog erstellt. Außerdem wird der Body (zumindest bei GET) in eine Datei geschrieben. Die Funktion "send_POST_PUT()" funktioniert im grunde genau wie das gezeigte Material mit einem unterschied in den Head-Feldern und das die Antwort nicht in einer Datei gespeichert wird.

Auch Fehler beim schreiben des Resultats in eine Datei werden abgefangen:

```
1try {
2    ofstream file;
3    file.open(input[7]);
4    file << body;
5    file.close();
6    spdlog::info(n + ". Request: Datei erfolgreich erstellt");
7} catch (...) {
```

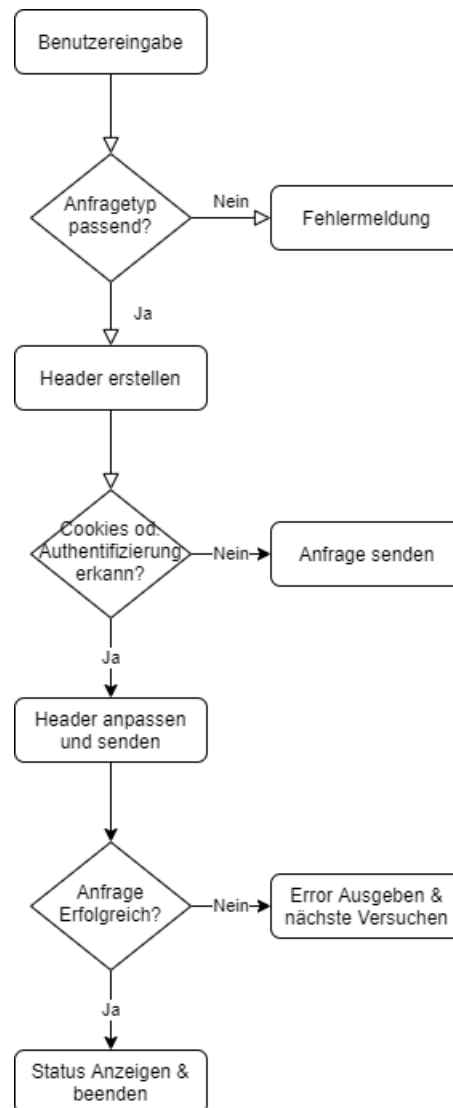
```

8  spdlog::error(n + ". Request: Datei erstellen fehlgeschlagen!");
9}

```

Falls ein Fehler auftritt wird der Benutzer wieder informiert.

Hier nochmal eine vereinfachte Zusammenfassung:



2.2.6 Schwierigkeiten

Die Lösung der Aufgabenstellung stellt den Programmierer vor einigen mehr oder weniger großen Problemen.

Zum einen muss man sich intensiv damit befassen für was welche Felder im HTTP-Header zuständig sind, ob sie zwingend notwendig sind und wann sie nicht gesetzt werden sollten. Z.B. habe ich anfangs aufgrund einer Fehlinterpretation das Feld "Connection" auf "keep-alive" gestellt. Das hat dazu geführt das die Ausführung des Programms sehr lange gedauert hat, da wie es der name eigentlich schon sagt die Verbindung offen gehalten wird. Das machte in diesem Fall aber keinen Sinn.

Auch die Kodierung der oben genannten Benutzer/Passwort-Kombination in Base64 bietet Fehlermöglichkeiten. Anfangs habe ich den Systemaufruf mit der Funktion "echo" in die Datei geschrieben. Da kam aber ine falscher Wert raus und die Authentifizierung schlug fehl. Mit "printf" wird es allerdings richtig Kodiert und der Fehler ist behoben.

Auch eine Möglichkeit mein Programm zu testen war schwer zu finden. Schließlich habe ich ptsv2.com entschieden. Diese Website bietet die Möglichkeit unter selbst benannten Verzeichnissen (in meinem Fall /t/jonny) GET und POST Anfragen zu senden. Auch eine Authentifizierung kann eingestellt werden. Somit ist dieses Tool fast perfekt für meine Zwecke. Die Anfragen können eingesehen und analysiert werden bezüglich Header, Cookies und Inhalt.

2.2.7 Bedienung

Für jeden zu sendenden Wert gibt es eine eigene Option. Ein Aufruf der 2 Anfragen sendet könnte so aussehen:

```
./klinger_project_2 --type1 GET --url1 ptsv2.com --path1 /t/jonny/post --port1 80
--cookies1 cookie=value --file1 test1.txt --user1 user --pw1 password --type2 POST
--url2 ptsv2.com --path2 /t/jonny/post --port2 80 --contentType2 text/plain
--content2 Content ...
```

Beachte: Bei POST und PUT wird kein Dateiname erwartet! Außerdem ist als Standard-Port 80 und Standard-Path / eingestellt. Mittels der Datei config.json die sich im selben Verzeichnis befinden muss, die Benutzer/Passwort Kombinationen für alle 3 Anfragen festlegen.

Eine gute Möglichkeit aufrufe zu testen ist für mich ptsv2.com. Diese Website dient als "Mülleimer" für GET und POST anfragen. Unter /t/jonny/post kann man die Anfragen senden und unter ptsv2.com/t/jonny einsehen. Diese Möglichkeit sollte nach stand 16.04.2021 noch einige Tage zur verfügung stehen.

3 Sources

- TCP/IP Programmierung von Prof. Kolousek
- HTTP-Skripten von Prof. Kolousek

- https://www.w3schools.com/tags/ref_httpmethods.aspx
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP
- <https://www.w3.org/Protocols/rfc2616/rfc2616.html>
- <https://tools.ietf.org/html/rfc2616>