HTTPipe

MILETIC Marko, 5CHIF - 14

2021

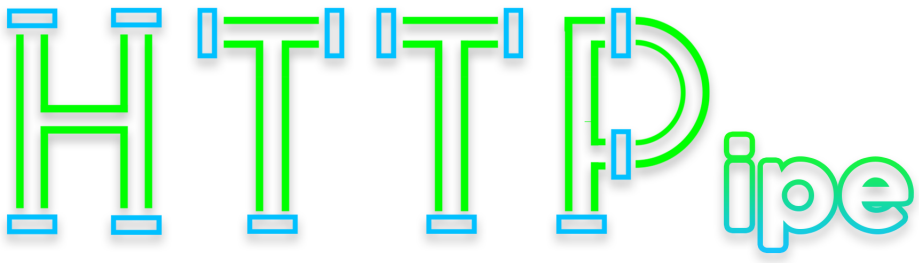

Figure 1: HTTPipe Logo

Contents

1 Introduction

Das Ziel dieses Projekts ist es einen HTTP 1.1 Client in C++ zu entwickeln. Der Name HTTPipe ergibt sich aus der Methode des pipen und, natürlich, aus HTTP.

Dieser Client soll möglichst benutzerfreundlich und leicht verstehbar sein.
Mithilfe des HTTPipe Clients lassen sich Dateien einfach per HTTP runterladen beziehungsweise hochladen.

Die Aufgabe bestand aus dem Senden einer Request und dem Abfangen und Verarbeiten einer Response.

2 Theoretische Grundlagen

Bevor ich auf die Impelementierung eingehe, möchte ich gerne etwas auf die ganze Thematik eingehen um das Verständnis aufzubereiten.

2.1 HTTP

HTTP steht für Hypertext Transfer Protocol“. Es wurde von Tim Berners-Lee am CERN (Schweiz) zusammen mit den anderen Konzepten entwickelt, die die Grundlagen fürs World Wide Web bilden: HTML und URI. Während HTML (Hypertext Markup Language) definiert, wie eine Webseite aufgebaut wird, legt die URL (Uniform Resource Locator) – eine Unterform des URI – fest, wie die Ressource (z. B. eine Webseite) im Web adressiert werden muss. HTTP hingegen regelt, wie diese Seite vom Server zum Client übertragen wird.

2.1.1 Wie funktioniert das?

Die Funktionsweise von HTTP lässt sich am einfachsten anhand des Aufrufs einer Webseite erläutern:

Der Nutzer tippt in die Adresszeile seines Internet-Browsers die URL http://example.com ein. Der Browser sendet eine entsprechende Anfrage, den HTTP-Request, an den zuständigen Webserver, der die Domäne example.com verwaltet.
Normalerweise lautet der Request: Sende mir bitte die Datei zu“. Alternativ kann der Client auch bloß fragen: Hast du diese Datei?“.
Der Webserver empfängt den HTTP-Request, sucht die gewünschte Datei und sendet als Erstes den Header, der dem anfragenden Client durch einen Status-Code das Resultat seiner Suche mitteilt.
Wenn die Datei gefunden wurde und der Client sie tatsächlich zugesendet haben will (und nicht nur wissen wollte, ob sie existiert), sendet der Server nach dem Header den Message Body, also den eigentlichen Inhalt und diesen kann der Client nun verarbeiten.

Die Frage die uns nun offen bleibt ist allerdings, was ist eine URL?

2.2 URL

Als URL (Abk.: Uniform Resource Locator) versteht man eine definierte Adresse, die auf die Position einer Datei auf einem Server zeigt und diese abruft. URLs werden in einem Webbrowser eingegeben, um auf Dokumente im Web zuzugreifen oder werden als Hyperlinks innerhalb eines Dokumentes eingebettet.

2.2.1 Bestandteile einer URL

Eine URL lässt sich in kleinere Teile aufteilen, diese kann man dann einzelnen Begriffen zuordnen.

- Protokoll-Präfix
- Portnummer (wird die Portnummer nicht angegeben, wir standardmäßig der Port 80 verwendet)
- Domain-Name oder IP-Adresse
- Namen der Unterverzeichnisse
- Dateiname

Daraus bildet sich eben die URL. Hier ein Beispiel:
http:// www.domainname.com/verzeichnis1/datei1.html

3 Struktur

In diesem Abschnitt wird die Architektur der Lösung mithilfe von Source Trail [2] abgebildet.

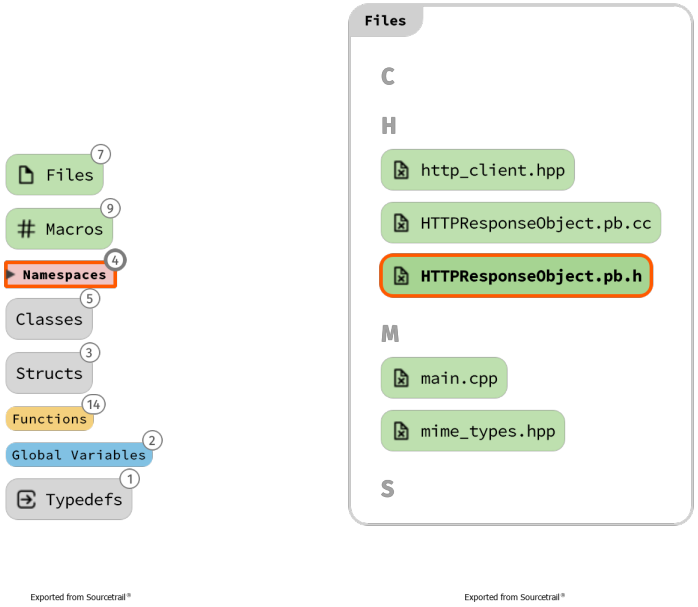

Figure 2: Overview

Als nächstes die Zusammenhänge

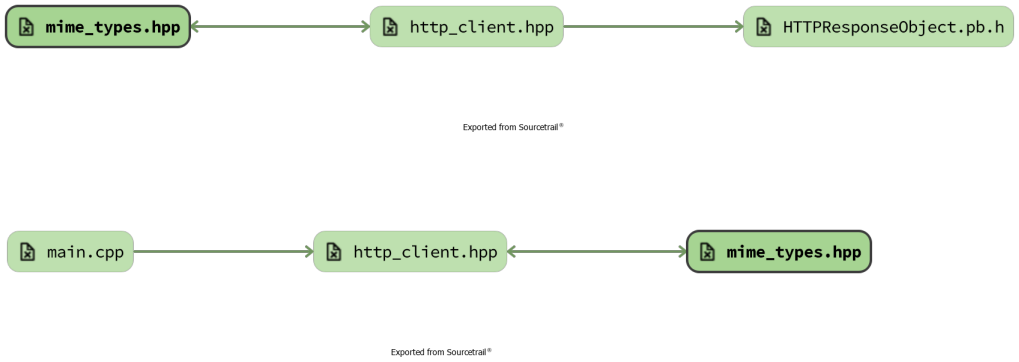

Figure 3: File-Relations

Hier sieht man den Aufbau der http_client.hpp Datei.

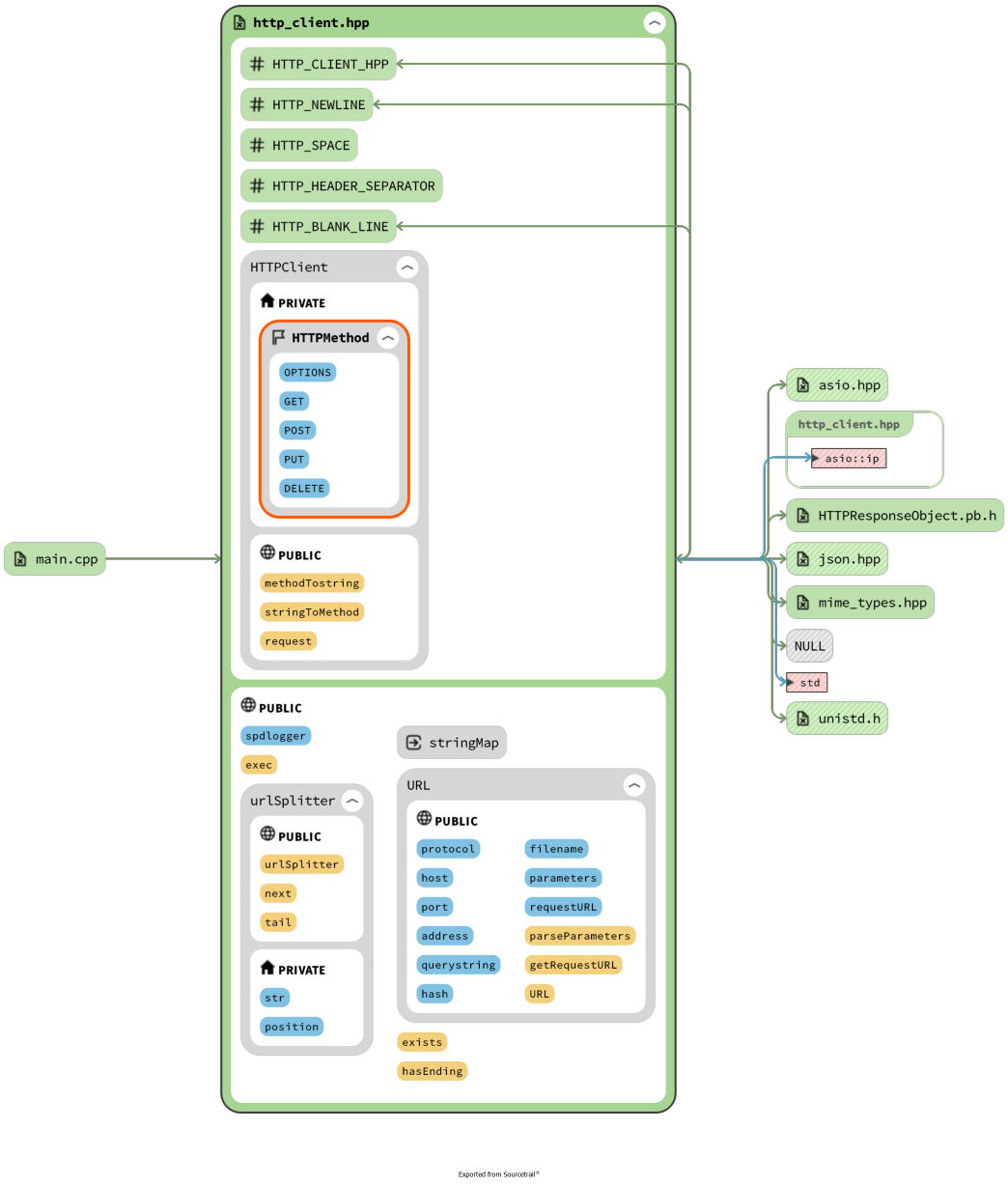

Figure 4: HTTPClient

Hier sieht man den Aufbau der mime_types.hpp Datei.

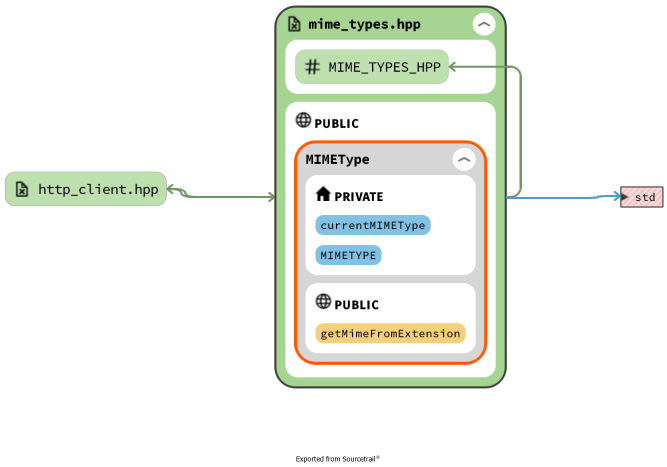

Figure 5: MIME Types

Und zu guter Letzt eine Auflistung der Klassen und Structs.

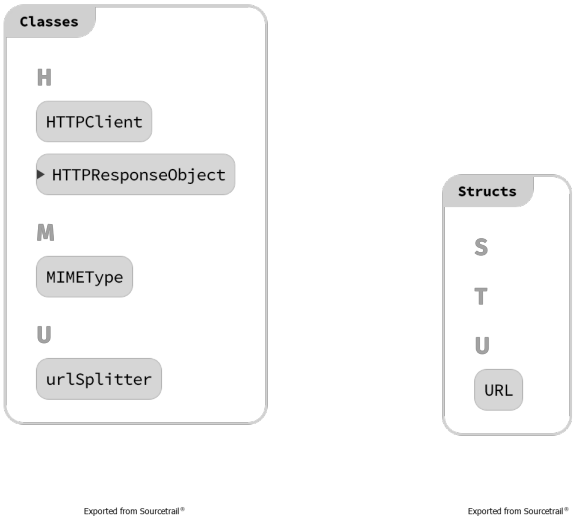

Figure 6: Classes and Structs

4 Implementierung

Was man vorab sagen kann ist das der einzige Schwachpunkt dieser Implementierungsart die vielen String-Operationen sind.
Dieses Projekt wurde in C++ realisiert.

4.1 HTTP-Client

Es wurde eine Klasse HTTPClient implementiert, welche die wichtigsten Methoden und Attribute enthält.

4.1.1 enum

Ein enum ist ein Datentyp für Variablen mit einer endlichen Wertemenge. Alle zulässigen Werte des enum's werden bei der Deklaration des Datentyps mit einem eindeutigen Namen (Identifikator) definiert, sie sind Symbole. Dabei kann auch eine Reihenfolge festgelegt werden, die eine Ordnung der einzelnen Werte bestimmt, nach der dann sortiert werden kann.
Bei enum's kann mitunter auch der textuelle Name eines Symbols direkt Verwendung finden, gewissermaßen als festgelegte Zeichenkette. [4]

```
// Type HTTPMethod => enum which contains the supported HTTP-
    Methods.
    typedef enum {
      OPTIONS = 0,
      GET,
      POST,
      PUT,
      DELETE
    } HTTPMethod;
```

4.2 Parsing Funktionen

Es wurden Funktionen erstellt, mit welchen die in 4.1.1 definierten Werte zu string geparsed werden können. Dies sollte auch andersrum funktionieren.

```
// Parses HTTPMethod to std::string (returns const char*).
    static const char *methodTostring(HTTPMethod method) {
        const char *methods[] = {"OPTIONS", "GET", "POST", "PUT",
                                 "DELETE", NULL};
        return methods[method];
    };

    // Parses std::string to HTTPMethod.
    static HTTPMethod stringToMethod(string method) {
        map<string, HTTPMethod> methods;

        methods["GET"] = HTTPMethod::GET;
        methods["POST"] = HTTPMethod::POST;
        methods["PUT"] = HTTPMethod::PUT;
        methods["DELETE"] = HTTPMethod::DELETE;

```

```
17          return methods[method];
18      };
```

4.3 Tokenizing

Um die URL besser verarbeiten zu können, wurde string Tokenizing verwendet. Das Tokenizing einer Zeichenfolge / eines strings bedeutet das Aufteilen einer Zeichenfolge in Bezug auf einige Trennzeichen oder Trenn-Kriterien. Hier wurde eben nach dem Aufbau einer URL Tokenizing angewendet.

```

/*
  @class: urlSplitter should be used for tokenizing the passed URL.


*/
class urlSplitter {
public:
  urlSplitter(string &str) : str(str), position(0){};

  string next(string search, bool returnTail = false) {
    size_t hit = str.find(search, position);
    if(hit == string::npos) {

      if(returnTail)
        return tail();
      else
        return "";

    }

    size_t oldPosition = position;
    position = hit + search.length();

    return str.substr(oldPosition, hit - oldPosition);
  };

  string tail() {
    size_t oldPosition = position;
    position = str.length();
    return str.substr(oldPosition);
  };

private:
  string str;
  size_t position;
};
```

4.4 URL Klasse

Nun kommt eine sehr wichtige Klasse, URL. Hier wurde im Konstruktor der Tokenizer - 4.3 - verwendet um die einzelnen Bestandteile der URL abspeichern zu können.

```
struct URL {

  public:
    string protocol, host, port,
          address, querystring, hash, filename;
```

```
    stringMap parameters;

    string requestURL = "";

    void parseParameters() {
      urlSplitter qt(querystring);
      do {
        string key = qt.next("=");

        if(key == "")
          break;

        parameters[key] = qt.next("&", true);
      } while (true);

    }

    string getRequestURL() {
      return requestURL;
    }




    /*
      @constructor:
              Splits the URL into smaller, understandable parts
              - is also able to parse params into a string map
    */
    URL(string input, bool shouldParseParameters = false) {

      //save the initial request URL to avoid redundant code.
      requestURL = input;


      // Create an instance of @class: urlSplitter.
      urlSplitter t = urlSplitter(input);

      // Get the protocol.
      protocol = t.next("://");

      // Get the host-port.
      string hostPortString = t.next("/");

      // Create another instance of @class: urlSplitter,
      //  which makes it easier to handle.
      urlSplitter hostPort(hostPortString);

      host = hostPort.next(hostPortString[0] == '[' ? "]:" : ":",
    true);

      if(host[0] == '[')
        host = host.substr(1, host.size() - 1);

      // Save the port.
      port = hostPort.tail();

```

```
62        // Get the address
63        address = t.next("?", true);
64
65        // Get (if existing) the name of the file.
66        if(address.find(".") < address.length()) {
67          filename = address.substr(address.find("/") + 1,
68                     address[address.length() - 1]);
69        }
70
71        // Save the query.
72        querystring = t.next("#", true);
73
74        // Get the hash.
75        hash = t.tail();
76
77        if(shouldParseParameters)
78          parseParameters();
79
80      };
81
82  };
```

4.5 Die Request

Nun kommen wir zum Hauptteil. Das Senden einer Request und das Verarbeiten der Response.
Hier ist die Funktion an sich. Diese ist relativ groß und hat viele Parameter. Hier könnte man durchaus noch mehr Funktionen für Teilaufgaben definieren. Allerdings wurde hier darauf Wert gelegt die Anzahl an Funktionsaufrufen zu minimieren.

```
1  static HTTPResponseObject::HTTPResponseObject
2          request(HTTPMethod method, URL URL,
3                  json json_data,
4                  string filePath = "",
5                  string auth_data = "",
6                  string save_dir = "", string cookie = "");
```

4.5.1 Vorgangsweise

Als erstes wird ein asio resolver erstellt, welcher die DNS-Arbeit für uns erledigt und danach wird eine Socket-Verbindung aufbaut. Als nächstes wird eine Request aufgebaut. (Nach den gegebenen Kriterien) Im folgenden Code-Snippet wird eine POST- oder PUT-Request aufgebaut und abgeschickt.[1]

```
// Create instance of @class: MIMETYPE
MIMEType mime;

// Determine which MIME-type the file has.
string mime_type = string(mime.getMimeFromExtension(filePath));

// Build a POST / PUT Request.

request_str = string(methodTostring(method)) + string(" /") +
            URL.address + ((URL.querystring == "") ? "" : "?") +
            URL.querystring + " HTTP/1.1" HTTP_NEWLINE
            "Host: " + URL.host + HTTP_NEWLINE
            "Content-Type: " + mime_type + HTTP_NEWLINE
            "Accept: */*" HTTP_NEWLINE
            "Authorization: Basic " + auth_data + HTTP_NEWLINE
            "Content-Length: " + to_string(file_length) +
    HTTP_NEWLINE
            "Connection: close" HTTP_NEWLINE
            "Cookie: " + cookie + HTTP_NEWLINE HTTP_NEWLINE;

request_stream << request_str;

// Send the request.
spdlog::get("http_client_logger")
->info("Sending...");

asio::write(socket, request);
```

[3]

Als nächstes wird die Response des Servers verarbeitet. Hier wird der Header immer in ein seperates Log File geschrieben.

```
// If a save directory was specified, add it to the file name
string fname = URL.filename;

// Create a new file to save the response in.
std::ofstream requested_file;
bool is_new_file = false;
bool is_log = false;

// Only save a log-file if the HTTPMethod is DELETE OR if no file
    was sent.
if(fname.compare("") == 0 || method == HTTPClient::DELETE) {
spdlog::get("http_client_logger")
    ->info("The response is being saved into a log-file");

fname = "../response_log/HTTPipe_log-" + timestamp + ".txt";
is_log = true;

```

```
} else {
    spdlog::get("http_client_logger")
        ->info("The response-file is being saved!");

    is_new_file = true;
}

// Always save the header.
if(save_dir.compare("") != 0 && is_log) {
    fname = save_dir + "/" + "HTTPipe_log-" + timestamp + ".txt";
    spdlog::get("http_client_logger")
        ->info("The response header is being saved into a log file"
    );

} else if(save_dir.compare("") != 0 && !is_log) {
    // Check if the path is ending with "/" or not
    if(hasEnding(save_dir, "/")) {
        fname = save_dir + URL.filename;
    } else {
        fname = save_dir + "/" + URL.filename;
    }


}
```

Jetzt wird gelesen bis ein End-Of-File Fehler geworfen wird.

```
// Write whatever content we already have to output.
if(response.size() > 0) {
    requested_file << &response;
}

// Read until EOF, writing data to output as we go.
while (asio::read(socket, response, asio::transfer_at_least(1))) {
    requested_file << &response;

}
```

Und das wars im Prinzip. Dieses Projekt wurde mit Liebe und Respekt von Marko Miletic entworfen und entwickelt.

Bibliography

[1] Non-boost asio.

[2] CoatiSoftware. Source trail.

[3] gabime. Spdlog.

[4] Wikipedia. Aufzählungstyp.