//

// httplib.h

//

// Copyright (c) 2019 Yuji Hirose. All rights reserved.

// MIT License

//

#ifndef CPPHTTPLIB\_HTTPLIB\_H

#define CPPHTTPLIB\_HTTPLIB\_H

/\*

\* Configuration

\*/

#ifndef CPPHTTPLIB\_KEEPALIVE\_TIMEOUT\_SECOND

#define CPPHTTPLIB\_KEEPALIVE\_TIMEOUT\_SECOND 5

#endif

#ifndef CPPHTTPLIB\_KEEPALIVE\_TIMEOUT\_USECOND

#define CPPHTTPLIB\_KEEPALIVE\_TIMEOUT\_USECOND 0

#endif

#ifndef CPPHTTPLIB\_KEEPALIVE\_MAX\_COUNT

#define CPPHTTPLIB\_KEEPALIVE\_MAX\_COUNT 5

#endif

#ifndef CPPHTTPLIB\_READ\_TIMEOUT\_SECOND

#define CPPHTTPLIB\_READ\_TIMEOUT\_SECOND 5

#endif

#ifndef CPPHTTPLIB\_READ\_TIMEOUT\_USECOND

#define CPPHTTPLIB\_READ\_TIMEOUT\_USECOND 0

#endif

#ifndef CPPHTTPLIB\_REQUEST\_URI\_MAX\_LENGTH

#define CPPHTTPLIB\_REQUEST\_URI\_MAX\_LENGTH 8192

#endif

#ifndef CPPHTTPLIB\_REDIRECT\_MAX\_COUNT

#define CPPHTTPLIB\_REDIRECT\_MAX\_COUNT 20

#endif

#ifndef CPPHTTPLIB\_PAYLOAD\_MAX\_LENGTH

#define CPPHTTPLIB\_PAYLOAD\_MAX\_LENGTH (std::numeric\_limits<size\_t>::max)()

#endif

#ifndef CPPHTTPLIB\_RECV\_BUFSIZ

#define CPPHTTPLIB\_RECV\_BUFSIZ size\_t(4096u)

#endif

#ifndef CPPHTTPLIB\_THREAD\_POOL\_COUNT

#define CPPHTTPLIB\_THREAD\_POOL\_COUNT 8

#endif

/\*

\* Headers

\*/

#ifdef \_WIN32

#ifndef \_CRT\_SECURE\_NO\_WARNINGS

#define \_CRT\_SECURE\_NO\_WARNINGS

#endif //\_CRT\_SECURE\_NO\_WARNINGS

#ifndef \_CRT\_NONSTDC\_NO\_DEPRECATE

#define \_CRT\_NONSTDC\_NO\_DEPRECATE

#endif //\_CRT\_NONSTDC\_NO\_DEPRECATE

#if defined(\_MSC\_VER)

#ifdef \_WIN64

using ssize\_t = \_\_int64;

#else

using ssize\_t = int;

#endif

#if \_MSC\_VER < 1900

#define snprintf \_snprintf\_s

#endif

#endif // \_MSC\_VER

#ifndef S\_ISREG

#define S\_ISREG(m) (((m)&S\_IFREG) == S\_IFREG)

#endif // S\_ISREG

#ifndef S\_ISDIR

#define S\_ISDIR(m) (((m)&S\_IFDIR) == S\_IFDIR)

#endif // S\_ISDIR

#ifndef NOMINMAX

#define NOMINMAX

#endif // NOMINMAX

#include <io.h>

#include <winsock2.h>

#include <ws2tcpip.h>

#ifndef WSA\_FLAG\_NO\_HANDLE\_INHERIT

#define WSA\_FLAG\_NO\_HANDLE\_INHERIT 0x80

#endif

#ifdef \_MSC\_VER

#pragma comment(lib, "ws2\_32.lib")

#endif

#ifndef strcasecmp

#define strcasecmp \_stricmp

#endif // strcasecmp

using socket\_t = SOCKET;

#ifdef CPPHTTPLIB\_USE\_POLL

#define poll(fds, nfds, timeout) WSAPoll(fds, nfds, timeout)

#endif

#else // not \_WIN32

#include <arpa/inet.h>

#include <cstring>

#include <netdb.h>

#include <netinet/in.h>

#ifdef CPPHTTPLIB\_USE\_POLL

#include <poll.h>

#endif

#include <pthread.h>

#include <csignal>

#include <sys/select.h>

#include <sys/socket.h>

#include <unistd.h>

using socket\_t = int;

#define INVALID\_SOCKET (-1)

#endif //\_WIN32

#include <array>

#include <atomic>

#include <cassert>

#include <condition\_variable>

#include <errno.h>

#include <fcntl.h>

#include <fstream>

#include <functional>

#include <list>

#include <map>

#include <memory>

#include <mutex>

#include <random>

#include <regex>

#include <string>

#include <sys/stat.h>

#include <thread>

#ifdef CPPHTTPLIB\_OPENSSL\_SUPPORT

#include <openssl/err.h>

#include <openssl/ssl.h>

#include <openssl/x509v3.h>

// #if OPENSSL\_VERSION\_NUMBER < 0x1010100fL

// #error Sorry, OpenSSL versions prior to 1.1.1 are not supported

// #endif

#if OPENSSL\_VERSION\_NUMBER < 0x10100000L

#include <openssl/crypto.h>

inline const unsigned char \*ASN1\_STRING\_get0\_data(const ASN1\_STRING \*asn1) {

return M\_ASN1\_STRING\_data(asn1);

}

#endif

#endif

#ifdef CPPHTTPLIB\_ZLIB\_SUPPORT

#include <zlib.h>

#endif

/\*

\* Declaration

\*/

namespace httplib {

namespace detail {

struct ci {

bool operator()(const std::string &s1, const std::string &s2) const {

return std::lexicographical\_compare(

s1.begin(), s1.end(), s2.begin(), s2.end(),

[](char c1, char c2) { return ::tolower(c1) < ::tolower(c2); });

}

};

} // namespace detail

enum class HttpVersion { v1\_0 = 0, v1\_1 };

using Headers = std::multimap<std::string, std::string, detail::ci>;

using Params = std::multimap<std::string, std::string>;

using Match = std::smatch;

using DataSink = std::function<void(const char \*data, size\_t data\_len)>;

using Done = std::function<void()>;

using ContentProvider = std::function<void(size\_t offset, size\_t length, DataSink sink)>;

using ContentProviderWithCloser = std::function<void(size\_t offset, size\_t length, DataSink sink, Done done)>;

using ContentReceiver = std::function<bool(const char \*data, size\_t data\_length)>;

using ContentReader = std::function<bool(ContentReceiver receiver)>;

using Progress = std::function<bool(uint64\_t current, uint64\_t total)>;

struct Response;

using ResponseHandler = std::function<bool(const Response &response)>;

struct MultipartFile {

std::string filename;

std::string content\_type;

size\_t offset = 0;

size\_t length = 0;

};

using MultipartFiles = std::multimap<std::string, MultipartFile>;

struct MultipartFormData {

std::string name;

std::string content;

std::string filename;

std::string content\_type;

};

using MultipartFormDataItems = std::vector<MultipartFormData>;

using Range = std::pair<ssize\_t, ssize\_t>;

using Ranges = std::vector<Range>;

struct Request {

std::string method;

std::string path;

Headers headers;

std::string body;

// for server

std::string version;

std::string target;

Params params;

MultipartFiles files;

Ranges ranges;

Match matches;

// for client

size\_t redirect\_count = CPPHTTPLIB\_REDIRECT\_MAX\_COUNT;

ResponseHandler response\_handler;

ContentReceiver content\_receiver;

Progress progress;

#ifdef CPPHTTPLIB\_OPENSSL\_SUPPORT

const SSL \*ssl;

#endif

bool has\_header(const char \*key) const;

std::string get\_header\_value(const char \*key, size\_t id = 0) const;

size\_t get\_header\_value\_count(const char \*key) const;

void set\_header(const char \*key, const char \*val);

void set\_header(const char \*key, const std::string &val);

bool has\_param(const char \*key) const;

std::string get\_param\_value(const char \*key, size\_t id = 0) const;

size\_t get\_param\_value\_count(const char \*key) const;

bool has\_file(const char \*key) const;

MultipartFile get\_file\_value(const char \*key) const;

// private members...

size\_t content\_length;

ContentProvider content\_provider;

};

struct Response {

std::string version;

int status;

Headers headers;

std::string body;

bool has\_header(const char \*key) const;

std::string get\_header\_value(const char \*key, size\_t id = 0) const;

size\_t get\_header\_value\_count(const char \*key) const;

void set\_header(const char \*key, const char \*val);

void set\_header(const char \*key, const std::string &val);

void set\_redirect(const char \*url);

void set\_content(const char \*s, size\_t n, const char \*content\_type);

void set\_content(const std::string &s, const char \*content\_type);

void set\_content\_provider(

size\_t length,

std::function<void(size\_t offset, size\_t length, DataSink sink)> provider,

std::function<void()> resource\_releaser = [] {});

void set\_chunked\_content\_provider(

std::function<void(size\_t offset, DataSink sink, Done done)> provider,

std::function<void()> resource\_releaser = [] {});

Response() : status(-1), content\_length(0) {}

~Response() {

if (content\_provider\_resource\_releaser) {

content\_provider\_resource\_releaser();

}

}

// private members...

size\_t content\_length;

ContentProviderWithCloser content\_provider;

std::function<void()> content\_provider\_resource\_releaser;

};

class Stream {

public:

virtual ~Stream() = default;

virtual int read(char \*ptr, size\_t size) = 0;

virtual int write(const char \*ptr, size\_t size1) = 0;

virtual int write(const char \*ptr) = 0;

virtual int write(const std::string &s) = 0;

virtual std::string get\_remote\_addr() const = 0;

template <typename... Args>

int write\_format(const char \*fmt, const Args &... args);

};

class SocketStream : public Stream {

public:

SocketStream(socket\_t sock, time\_t read\_timeout\_sec,

time\_t read\_timeout\_usec);

~SocketStream() override;

int read(char \*ptr, size\_t size) override;

int write(const char \*ptr, size\_t size) override;

int write(const char \*ptr) override;

int write(const std::string &s) override;

std::string get\_remote\_addr() const override;

private:

socket\_t sock\_;

time\_t read\_timeout\_sec\_;

time\_t read\_timeout\_usec\_;

};

class BufferStream : public Stream {

public:

BufferStream() = default;

~BufferStream() override = default;

int read(char \*ptr, size\_t size) override;

int write(const char \*ptr, size\_t size) override;

int write(const char \*ptr) override;

int write(const std::string &s) override;

std::string get\_remote\_addr() const override;

const std::string &get\_buffer() const;

private:

std::string buffer;

};

class TaskQueue {

public:

TaskQueue() = default;

virtual ~TaskQueue() = default;

virtual void enqueue(std::function<void()> fn) = 0;

virtual void shutdown() = 0;

};

#if CPPHTTPLIB\_THREAD\_POOL\_COUNT > 0

class ThreadPool : public TaskQueue {

public:

explicit ThreadPool(size\_t n) : shutdown\_(false) {

while (n) {

threads\_.emplace\_back(worker(\*this));

n--;

}

}

ThreadPool(const ThreadPool &) = delete;

~ThreadPool() override = default;

void enqueue(std::function<void()> fn) override {

std::unique\_lock<std::mutex> lock(mutex\_);

jobs\_.push\_back(fn);

cond\_.notify\_one();

}

void shutdown() override {

// Stop all worker threads...

{

std::unique\_lock<std::mutex> lock(mutex\_);

shutdown\_ = true;

}

cond\_.notify\_all();

// Join...

for (auto& t : threads\_) {

t.join();

}

}

private:

struct worker {

explicit worker(ThreadPool &pool) : pool\_(pool) {}

void operator()() {

for (;;) {

std::function<void()> fn;

{

std::unique\_lock<std::mutex> lock(pool\_.mutex\_);

pool\_.cond\_.wait(

lock, [&] { return !pool\_.jobs\_.empty() || pool\_.shutdown\_; });

if (pool\_.shutdown\_ && pool\_.jobs\_.empty()) { break; }

fn = pool\_.jobs\_.front();

pool\_.jobs\_.pop\_front();

}

assert(true == static\_cast<bool>(fn));

fn();

}

}

ThreadPool &pool\_;

};

friend struct worker;

std::vector<std::thread> threads\_;

std::list<std::function<void()>> jobs\_;

bool shutdown\_;

std::condition\_variable cond\_;

std::mutex mutex\_;

};

#elif CPPHTTPLIB\_THREAD\_POOL\_COUNT == 0

class Threads : public TaskQueue {

public:

Threads() : running\_threads\_(0) {}

virtual ~Threads() {}

virtual void enqueue(std::function<void()> fn) override {

std::thread([=]() {

{

std::lock\_guard<std::mutex> guard(running\_threads\_mutex\_);

running\_threads\_++;

}

fn();

{

std::lock\_guard<std::mutex> guard(running\_threads\_mutex\_);

running\_threads\_--;

}

}).detach();

}

virtual void shutdown() override {

for (;;) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

std::lock\_guard<std::mutex> guard(running\_threads\_mutex\_);

if (!running\_threads\_) { break; }

}

}

private:

std::mutex running\_threads\_mutex\_;

int running\_threads\_;

};

#else

class NoThread : public TaskQueue {

public:

NoThread() {}

virtual ~NoThread() {}

virtual void enqueue(std::function<void()> fn) override {

fn();

}

virtual void shutdown() override {

}

};

#endif

class Server {

public:

using Handler = std::function<void(const Request &, Response &)>;

using HandlerWithContentReader = std::function<void(const Request &, Response &,

const ContentReader &content\_reader)>;

using Logger = std::function<void(const Request &, const Response &)>;

Server();

virtual ~Server();

virtual bool is\_valid() const;

Server &Get(const char \*pattern, Handler handler);

Server &Post(const char \*pattern, Handler handler);

Server &Post(const char \*pattern, HandlerWithContentReader handler);

Server &Put(const char \*pattern, Handler handler);

Server &Put(const char \*pattern, HandlerWithContentReader handler);

Server &Patch(const char \*pattern, Handler handler);

Server &Patch(const char \*pattern, HandlerWithContentReader handler);

Server &Delete(const char \*pattern, Handler handler);

Server &Options(const char \*pattern, Handler handler);

bool set\_base\_dir(const char \*dir, const char \*mount\_point = nullptr);

void set\_file\_request\_handler(Handler handler);

void set\_error\_handler(Handler handler);

void set\_logger(Logger logger);

void set\_keep\_alive\_max\_count(size\_t count);

void set\_read\_timeout(time\_t sec, time\_t usec);

void set\_payload\_max\_length(size\_t length);

bool bind\_to\_port(const char \*host, int port, int socket\_flags = 0);

int bind\_to\_any\_port(const char \*host, int socket\_flags = 0);

bool listen\_after\_bind();

bool listen(const char \*host, int port, int socket\_flags = 0);

bool is\_running() const;

void stop();

std::function<TaskQueue \*(void)> new\_task\_queue;

protected:

bool process\_request(Stream &strm, bool last\_connection,

bool &connection\_close,

const std::function<void(Request &)>& setup\_request);

size\_t keep\_alive\_max\_count\_;

time\_t read\_timeout\_sec\_;

time\_t read\_timeout\_usec\_;

size\_t payload\_max\_length\_;

private:

using Handlers = std::vector<std::pair<std::regex, Handler>>;

using HandersForContentReader = std::vector<std::pair<std::regex, HandlerWithContentReader>>;

socket\_t create\_server\_socket(const char \*host, int port,

int socket\_flags) const;

int bind\_internal(const char \*host, int port, int socket\_flags);

bool listen\_internal();

bool routing(Request &req, Response &res, Stream &strm, bool last\_connection);

bool handle\_file\_request(Request &req, Response &res);

bool dispatch\_request(Request &req, Response &res, Handlers &handlers);

bool dispatch\_request\_for\_content\_reader(Request &req, Response &res,

ContentReader content\_reader,

HandersForContentReader &handlers);

bool parse\_request\_line(const char \*s, Request &req);

bool write\_response(Stream &strm, bool last\_connection, const Request &req,

Response &res);

bool write\_content\_with\_provider(Stream &strm, const Request &req,

Response &res, const std::string &boundary,

const std::string &content\_type);

bool read\_content(Stream &strm, bool last\_connection, Request &req,

Response &res);

bool read\_content\_with\_content\_receiver(Stream &strm, bool last\_connection,

Request &req, Response &res,

ContentReceiver reveiver);

virtual bool process\_and\_close\_socket(socket\_t sock);

std::atomic<bool> is\_running\_;

std::atomic<socket\_t> svr\_sock\_;

std::vector<std::pair<std::string, std::string>> base\_dirs\_;

Handler file\_request\_handler\_;

Handlers get\_handlers\_;

Handlers post\_handlers\_;

HandersForContentReader post\_handlers\_for\_content\_reader;

Handlers put\_handlers\_;

HandersForContentReader put\_handlers\_for\_content\_reader;

Handlers patch\_handlers\_;

HandersForContentReader patch\_handlers\_for\_content\_reader;

Handlers delete\_handlers\_;

Handlers options\_handlers\_;

Handler error\_handler\_;

Logger logger\_;

};

class Client {

public:

explicit Client(const char \*host, int port = 80, time\_t timeout\_sec = 300);

virtual ~Client();

virtual bool is\_valid() const;

std::shared\_ptr<Response> Get(const char \*path);

std::shared\_ptr<Response> Get(const char \*path, const Headers &headers);

std::shared\_ptr<Response> Get(const char \*path, Progress progress);

std::shared\_ptr<Response> Get(const char \*path, const Headers &headers,

Progress progress);

std::shared\_ptr<Response> Get(const char \*path,

ContentReceiver content\_receiver);

std::shared\_ptr<Response> Get(const char \*path, const Headers &headers,

ContentReceiver content\_receiver);

std::shared\_ptr<Response>

Get(const char \*path, ContentReceiver content\_receiver, Progress progress);

std::shared\_ptr<Response> Get(const char \*path, const Headers &headers,

ContentReceiver content\_receiver,

Progress progress);

std::shared\_ptr<Response> Get(const char \*path, const Headers &headers,

ResponseHandler response\_handler,

ContentReceiver content\_receiver);

std::shared\_ptr<Response> Get(const char \*path, const Headers &headers,

ResponseHandler response\_handler,

ContentReceiver content\_receiver,

Progress progress);

std::shared\_ptr<Response> Head(const char \*path);

std::shared\_ptr<Response> Head(const char \*path, const Headers &headers);

std::shared\_ptr<Response> Post(const char \*path, const std::string &body,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Post(const char \*path, const Headers &headers,

const std::string &body,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Post(const char \*path, size\_t content\_length,

ContentProvider content\_provider,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Post(const char \*path, const Headers &headers,

size\_t content\_length,

ContentProvider content\_provider,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Post(const char \*path, const Params &params,

bool compress = false);

std::shared\_ptr<Response> Post(const char \*path, const Headers &headers,

const Params &params, bool compress = false);

std::shared\_ptr<Response> Post(const char \*path,

const MultipartFormDataItems &items,

bool compress = false);

std::shared\_ptr<Response> Post(const char \*path, const Headers &headers,

const MultipartFormDataItems &items,

bool compress = false);

std::shared\_ptr<Response> Put(const char \*path, const std::string &body,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Put(const char \*path, const Headers &headers,

const std::string &body,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Put(const char \*path, size\_t content\_length,

ContentProvider content\_provider,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Put(const char \*path, const Headers &headers,

size\_t content\_length,

ContentProvider content\_provider,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Patch(const char \*path, const std::string &body,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Patch(const char \*path, const Headers &headers,

const std::string &body,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Patch(const char \*path, size\_t content\_length,

ContentProvider content\_provider,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Patch(const char \*path, const Headers &headers,

size\_t content\_length,

ContentProvider content\_provider,

const char \*content\_type,

bool compress = false);

std::shared\_ptr<Response> Delete(const char \*path);

std::shared\_ptr<Response> Delete(const char \*path, const std::string &body,

const char \*content\_type);

std::shared\_ptr<Response> Delete(const char \*path, const Headers &headers);

std::shared\_ptr<Response> Delete(const char \*path, const Headers &headers,

const std::string &body,

const char \*content\_type);

std::shared\_ptr<Response> Options(const char \*path);

std::shared\_ptr<Response> Options(const char \*path, const Headers &headers);

bool send(const Request &req, Response &res);

bool send(const std::vector<Request> &requests,

std::vector<Response> &responses);

void set\_keep\_alive\_max\_count(size\_t count);

void set\_read\_timeout(time\_t sec, time\_t usec);

void follow\_location(bool on);

protected:

bool process\_request(Stream &strm, const Request &req, Response &res,

bool last\_connection, bool &connection\_close);

const std::string host\_;

const int port\_;

time\_t timeout\_sec\_;

const std::string host\_and\_port\_;

size\_t keep\_alive\_max\_count\_;

time\_t read\_timeout\_sec\_;

time\_t read\_timeout\_usec\_;

size\_t follow\_location\_;

private:

socket\_t create\_client\_socket() const;

bool read\_response\_line(Stream &strm, Response &res);

void write\_request(Stream &strm, const Request &req, bool last\_connection);

bool redirect(const Request &req, Response &res);

std::shared\_ptr<Response>

send\_with\_content\_provider(const char \*method, const char \*path,

const Headers &headers, const std::string &body,

size\_t content\_length,

ContentProvider content\_provider,

const char \*content\_type, bool compress);

virtual bool process\_and\_close\_socket(

socket\_t sock, size\_t request\_count,

std::function<bool(Stream &strm, bool last\_connection,

bool &connection\_close)>

callback);

virtual bool is\_ssl() const;

};

inline void Get(std::vector<Request> &requests, const char \*path,

const Headers &headers) {

Request req;

req.method = "GET";

req.path = path;

req.headers = headers;

requests.emplace\_back(std::move(req));

}

inline void Get(std::vector<Request> &requests, const char \*path) {

Get(requests, path, Headers());

}

inline void Post(std::vector<Request> &requests, const char \*path,

const Headers &headers, const std::string &body,

const char \*content\_type) {

Request req;

req.method = "POST";

req.path = path;

req.headers = headers;

req.headers.emplace("Content-Type", content\_type);

req.body = body;

requests.emplace\_back(std::move(req));

}

inline void Post(std::vector<Request> &requests, const char \*path,

const std::string &body, const char \*content\_type) {

Post(requests, path, Headers(), body, content\_type);

}

#ifdef CPPHTTPLIB\_OPENSSL\_SUPPORT

class SSLSocketStream : public Stream {

public:

SSLSocketStream(socket\_t sock, SSL \*ssl, time\_t read\_timeout\_sec,

time\_t read\_timeout\_usec);

virtual ~SSLSocketStream();

virtual int read(char \*ptr, size\_t size);

virtual int write(const char \*ptr, size\_t size);

virtual int write(const char \*ptr);

virtual int write(const std::string &s);

virtual std::string get\_remote\_addr() const;

private:

socket\_t sock\_;

SSL \*ssl\_;

time\_t read\_timeout\_sec\_;

time\_t read\_timeout\_usec\_;

};

class SSLServer : public Server {

public:

SSLServer(const char \*cert\_path, const char \*private\_key\_path,

const char \*client\_ca\_cert\_file\_path = nullptr,

const char \*client\_ca\_cert\_dir\_path = nullptr);

virtual ~SSLServer();

virtual bool is\_valid() const;

private:

virtual bool process\_and\_close\_socket(socket\_t sock);

SSL\_CTX \*ctx\_;

std::mutex ctx\_mutex\_;

};

class SSLClient : public Client {

public:

SSLClient(const char \*host, int port = 443, time\_t timeout\_sec = 300,

const char \*client\_cert\_path = nullptr,

const char \*client\_key\_path = nullptr);

virtual ~SSLClient();

virtual bool is\_valid() const;

void set\_ca\_cert\_path(const char \*ca\_ceert\_file\_path,

const char \*ca\_cert\_dir\_path = nullptr);

void enable\_server\_certificate\_verification(bool enabled);

long get\_openssl\_verify\_result() const;

SSL\_CTX \*ssl\_context() const noexcept;

private:

virtual bool process\_and\_close\_socket(

socket\_t sock, size\_t request\_count,

std::function<bool(Stream &strm, bool last\_connection,

bool &connection\_close)>

callback);

virtual bool is\_ssl() const;

bool verify\_host(X509 \*server\_cert) const;

bool verify\_host\_with\_subject\_alt\_name(X509 \*server\_cert) const;

bool verify\_host\_with\_common\_name(X509 \*server\_cert) const;

bool check\_host\_name(const char \*pattern, size\_t pattern\_len) const;

SSL\_CTX \*ctx\_;

std::mutex ctx\_mutex\_;

std::vector<std::string> host\_components\_;

std::string ca\_cert\_file\_path\_;

std::string ca\_cert\_dir\_path\_;

bool server\_certificate\_verification\_ = false;

long verify\_result\_ = 0;

};

#endif

/\*

\* Implementation

\*/

namespace detail {

inline bool is\_hex(char c, int &v) {

if (0x20 <= c && isdigit(c)) {

v = c - '0';

return true;

} else if ('A' <= c && c <= 'F') {

v = c - 'A' + 10;

return true;

} else if ('a' <= c && c <= 'f') {

v = c - 'a' + 10;

return true;

}

return false;

}

inline bool from\_hex\_to\_i(const std::string &s, size\_t i, size\_t cnt,

int &val) {

if (i >= s.size()) { return false; }

val = 0;

for (; cnt; i++, cnt--) {

if (!s[i]) { return false; }

int v = 0;

if (is\_hex(s[i], v)) {

val = val \* 16 + v;

} else {

return false;

}

}

return true;

}

inline std::string from\_i\_to\_hex(size\_t n) {

const char \*charset = "0123456789abcdef";

std::string ret;

do {

ret = charset[n & 15] + ret;

n >>= 4;

} while (n > 0);

return ret;

}

inline size\_t to\_utf8(int code, char \*buff) {

if (code < 0x0080) {

buff[0] = (code & 0x7F);

return 1;

} else if (code < 0x0800) {

buff[0] = (0xC0 | ((code >> 6) & 0x1F));

buff[1] = (0x80 | (code & 0x3F));

return 2;

} else if (code < 0xD800) {

buff[0] = (0xE0 | ((code >> 12) & 0xF));

buff[1] = (0x80 | ((code >> 6) & 0x3F));

buff[2] = (0x80 | (code & 0x3F));

return 3;

} else if (code < 0xE000) { // D800 - DFFF is invalid...

return 0;

} else if (code < 0x10000) {

buff[0] = (0xE0 | ((code >> 12) & 0xF));

buff[1] = (0x80 | ((code >> 6) & 0x3F));

buff[2] = (0x80 | (code & 0x3F));

return 3;

} else if (code < 0x110000) {

buff[0] = (0xF0 | ((code >> 18) & 0x7));

buff[1] = (0x80 | ((code >> 12) & 0x3F));

buff[2] = (0x80 | ((code >> 6) & 0x3F));

buff[3] = (0x80 | (code & 0x3F));

return 4;

}

// NOTREACHED

return 0;

}

// NOTE: This code came up with the following stackoverflow post:

// https://stackoverflow.com/questions/180947/base64-decode-snippet-in-c

inline std::string base64\_encode(const std::string &in) {

static const auto lookup =

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

std::string out;

out.reserve(in.size());

int val = 0;

int valb = -6;

for (uint8\_t c : in) {

val = (val << 8) + c;

valb += 8;

while (valb >= 0) {

out.push\_back(lookup[(val >> valb) & 0x3F]);

valb -= 6;

}

}

if (valb > -6) { out.push\_back(lookup[((val << 8) >> (valb + 8)) & 0x3F]); }

while (out.size() % 4) {

out.push\_back('=');

}

return out;

}

inline bool is\_file(const std::string &path) {

struct stat st;

return stat(path.c\_str(), &st) >= 0 && S\_ISREG(st.st\_mode);

}

inline bool is\_dir(const std::string &path) {

struct stat st;

return stat(path.c\_str(), &st) >= 0 && S\_ISDIR(st.st\_mode);

}

inline bool is\_valid\_path(const std::string &path) {

size\_t level = 0;

size\_t i = 0;

// Skip slash

while (i < path.size() && path[i] == '/') {

i++;

}

while (i < path.size()) {

// Read component

auto beg = i;

while (i < path.size() && path[i] != '/') {

i++;

}

auto len = i - beg;

assert(len > 0);

if (!path.compare(beg, len, ".")) {

;

} else if (!path.compare(beg, len, "..")) {

if (level == 0) { return false; }

level--;

} else {

level++;

}

// Skip slash

while (i < path.size() && path[i] == '/') {

i++;

}

}

return true;

}

inline void read\_file(const std::string &path, std::string &out) {

std::ifstream fs(path, std::ios\_base::binary);

fs.seekg(0, std::ios\_base::end);

auto size = fs.tellg();

fs.seekg(0);

out.resize(static\_cast<size\_t>(size));

fs.read(&out[0], size);

}

inline std::string file\_extension(const std::string &path) {

std::smatch m;

auto re = std::regex("\\.([a-zA-Z0-9]+)$");

if (std::regex\_search(path, m, re)) { return m[1].str(); }

return std::string();

}

template <class Fn> void split(const char \*b, const char \*e, char d, Fn fn) {

int i = 0;

int beg = 0;

while (e ? (b + i != e) : (b[i] != '\0')) {

if (b[i] == d) {

fn(&b[beg], &b[i]);

beg = i + 1;

}

i++;

}

if (i) { fn(&b[beg], &b[i]); }

}

// NOTE: until the read size reaches `fixed\_buffer\_size`, use `fixed\_buffer`

// to store data. The call can set memory on stack for performance.

class stream\_line\_reader {

public:

stream\_line\_reader(Stream &strm, char \*fixed\_buffer, size\_t fixed\_buffer\_size)

: strm\_(strm), fixed\_buffer\_(fixed\_buffer),

fixed\_buffer\_size\_(fixed\_buffer\_size) {}

const char \*ptr() const {

if (glowable\_buffer\_.empty()) {

return fixed\_buffer\_;

} else {

return glowable\_buffer\_.data();

}

}

size\_t size() const {

if (glowable\_buffer\_.empty()) {

return fixed\_buffer\_used\_size\_;

} else {

return glowable\_buffer\_.size();

}

}

bool getline() {

fixed\_buffer\_used\_size\_ = 0;

glowable\_buffer\_.clear();

for (size\_t i = 0;; i++) {

char byte;

auto n = strm\_.read(&byte, 1);

if (n < 0) {

return false;

} else if (n == 0) {

if (i == 0) {

return false;

} else {

break;

}

}

append(byte);

if (byte == '\n') { break; }

}

return true;

}

private:

void append(char c) {

if (fixed\_buffer\_used\_size\_ < fixed\_buffer\_size\_ - 1) {

fixed\_buffer\_[fixed\_buffer\_used\_size\_++] = c;

fixed\_buffer\_[fixed\_buffer\_used\_size\_] = '\0';

} else {

if (glowable\_buffer\_.empty()) {

assert(fixed\_buffer\_[fixed\_buffer\_used\_size\_] == '\0');

glowable\_buffer\_.assign(fixed\_buffer\_, fixed\_buffer\_used\_size\_);

}

glowable\_buffer\_ += c;

}

}

Stream &strm\_;

char \*fixed\_buffer\_;

const size\_t fixed\_buffer\_size\_;

size\_t fixed\_buffer\_used\_size\_ = 0;

std::string glowable\_buffer\_;

};

inline int close\_socket(socket\_t sock) {

#ifdef \_WIN32

return closesocket(sock);

#else

return close(sock);

#endif

}

inline int select\_read(socket\_t sock, time\_t sec, time\_t usec) {

#ifdef CPPHTTPLIB\_USE\_POLL

struct pollfd pfd\_read;

pfd\_read.fd = sock;

pfd\_read.events = POLLIN;

auto timeout = static\_cast<int>(sec \* 1000 + usec / 1000);

return poll(&pfd\_read, 1, timeout);

#else

fd\_set fds;

FD\_ZERO(&fds);

FD\_SET(sock, &fds);

timeval tv;

tv.tv\_sec = static\_cast<long>(sec);

tv.tv\_usec = static\_cast<long>(usec);

return select(static\_cast<int>(sock + 1), &fds, nullptr, nullptr, &tv);

#endif

}

inline bool wait\_until\_socket\_is\_ready(socket\_t sock, time\_t sec, time\_t usec) {

#ifdef CPPHTTPLIB\_USE\_POLL

struct pollfd pfd\_read;

pfd\_read.fd = sock;

pfd\_read.events = POLLIN | POLLOUT;

auto timeout = static\_cast<int>(sec \* 1000 + usec / 1000);

if (poll(&pfd\_read, 1, timeout) > 0 &&

pfd\_read.revents & (POLLIN | POLLOUT)) {

int error = 0;

socklen\_t len = sizeof(error);

return getsockopt(sock, SOL\_SOCKET, SO\_ERROR,

reinterpret\_cast<char \*>(&error), &len) >= 0 &&

!error;

}

return false;

#else

fd\_set fdsr;

FD\_ZERO(&fdsr);

FD\_SET(sock, &fdsr);

auto fdsw = fdsr;

auto fdse = fdsr;

timeval tv;

tv.tv\_sec = static\_cast<long>(sec);

tv.tv\_usec = static\_cast<long>(usec);

if (select(static\_cast<int>(sock + 1), &fdsr, &fdsw, &fdse, &tv) > 0 &&

(FD\_ISSET(sock, &fdsr) || FD\_ISSET(sock, &fdsw))) {

int error = 0;

socklen\_t len = sizeof(error);

return getsockopt(sock, SOL\_SOCKET, SO\_ERROR, reinterpret\_cast<char\*>(&error), &len) >= 0 &&

!error;

}

return false;

#endif

}

template <typename T>

inline bool process\_and\_close\_socket(bool is\_client\_request, socket\_t sock,

size\_t keep\_alive\_max\_count,

time\_t read\_timeout\_sec,

time\_t read\_timeout\_usec, T callback) {

assert(keep\_alive\_max\_count > 0);

bool ret = false;

if (keep\_alive\_max\_count > 1) {

auto count = keep\_alive\_max\_count;

while (count > 0 &&

(is\_client\_request ||

detail::select\_read(sock, CPPHTTPLIB\_KEEPALIVE\_TIMEOUT\_SECOND,

CPPHTTPLIB\_KEEPALIVE\_TIMEOUT\_USECOND) > 0)) {

SocketStream strm(sock, read\_timeout\_sec, read\_timeout\_usec);

auto last\_connection = count == 1;

auto connection\_close = false;

ret = callback(strm, last\_connection, connection\_close);

if (!ret || connection\_close) { break; }

count--;

}

} else {

SocketStream strm(sock, read\_timeout\_sec, read\_timeout\_usec);

auto dummy\_connection\_close = false;

ret = callback(strm, true, dummy\_connection\_close);

}

close\_socket(sock);

return ret;

}

inline int shutdown\_socket(socket\_t sock) {

#ifdef \_WIN32

return shutdown(sock, SD\_BOTH);

#else

return shutdown(sock, SHUT\_RDWR);

#endif

}

template <typename Fn>

socket\_t create\_socket(const char \*host, int port, Fn fn,

int socket\_flags = 0) {

#ifdef \_WIN32

#define SO\_SYNCHRONOUS\_NONALERT 0x20

#define SO\_OPENTYPE 0x7008

int opt = SO\_SYNCHRONOUS\_NONALERT;

setsockopt(INVALID\_SOCKET, SOL\_SOCKET, SO\_OPENTYPE, (char \*)&opt,

sizeof(opt));

#endif

// Get address info

struct addrinfo hints;

struct addrinfo \*result;

memset(&hints, 0, sizeof(struct addrinfo));

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_flags = socket\_flags;

hints.ai\_protocol = 0;

auto service = std::to\_string(port);

if (getaddrinfo(host, service.c\_str(), &hints, &result)) {

return INVALID\_SOCKET;

}

for (auto rp = result; rp; rp = rp->ai\_next) {

// Create a socket

#ifdef \_WIN32

auto sock = WSASocketW(rp->ai\_family, rp->ai\_socktype, rp->ai\_protocol,

nullptr, 0, WSA\_FLAG\_NO\_HANDLE\_INHERIT);

#else

auto sock = socket(rp->ai\_family, rp->ai\_socktype, rp->ai\_protocol);

#endif

if (sock == INVALID\_SOCKET) { continue; }

#ifndef \_WIN32

if (fcntl(sock, F\_SETFD, FD\_CLOEXEC) == -1) { continue; }

#endif

// Make 'reuse address' option available

int yes = 1;

setsockopt(sock, SOL\_SOCKET, SO\_REUSEADDR, reinterpret\_cast<char \*>(&yes),

sizeof(yes));

#ifdef SO\_REUSEPORT

setsockopt(sock, SOL\_SOCKET, SO\_REUSEPORT, reinterpret\_cast<char \*>(&yes),

sizeof(yes));

#endif

// bind or connect

if (fn(sock, \*rp)) {

freeaddrinfo(result);

return sock;

}

close\_socket(sock);

}

freeaddrinfo(result);

return INVALID\_SOCKET;

}

inline void set\_nonblocking(socket\_t sock, bool nonblocking) {

#ifdef \_WIN32

auto flags = nonblocking ? 1UL : 0UL;

ioctlsocket(sock, FIONBIO, &flags);

#else

auto flags = fcntl(sock, F\_GETFL, 0);

fcntl(sock, F\_SETFL,

nonblocking ? (flags | O\_NONBLOCK) : (flags & (~O\_NONBLOCK)));

#endif

}

inline bool is\_connection\_error() {

#ifdef \_WIN32

return WSAGetLastError() != WSAEWOULDBLOCK;

#else

return errno != EINPROGRESS;

#endif

}

inline std::string get\_remote\_addr(socket\_t sock) {

struct sockaddr\_storage addr;

socklen\_t len = sizeof(addr);

if (!getpeername(sock, reinterpret\_cast<struct sockaddr \*>(&addr), &len)) {

std::array<char, NI\_MAXHOST> ipstr{};

if (!getnameinfo(reinterpret\_cast<struct sockaddr \*>(&addr), len, ipstr.data(), ipstr.size(),

nullptr, 0, NI\_NUMERICHOST)) {

return ipstr.data();

}

}

return std::string();

}

inline const char \*find\_content\_type(const std::string &path) {

auto ext = file\_extension(path);

if (ext == "txt") {

return "text/plain";

} else if (ext == "html" || ext == "htm") {

return "text/html";

} else if (ext == "css") {

return "text/css";

} else if (ext == "jpeg" || ext == "jpg") {

return "image/jpg";

} else if (ext == "png") {

return "image/png";

} else if (ext == "gif") {

return "image/gif";

} else if (ext == "svg") {

return "image/svg+xml";

} else if (ext == "ico") {

return "image/x-icon";

} else if (ext == "json") {

return "application/json";

} else if (ext == "pdf") {

return "application/pdf";

} else if (ext == "js") {

return "application/javascript";

} else if (ext == "xml") {

return "application/xml";

} else if (ext == "xhtml") {

return "application/xhtml+xml";

}

return nullptr;

}

inline const char \*status\_message(int status) {

switch (status) {

case 200: return "OK";

case 206: return "Partial Content";

case 301: return "Moved Permanently";

case 302: return "Found";

case 303: return "See Other";

case 304: return "Not Modified";

case 400: return "Bad Request";

case 403: return "Forbidden";

case 404: return "Not Found";

case 413: return "Payload Too Large";

case 414: return "Request-URI Too Long";

case 415: return "Unsupported Media Type";

case 416: return "Range Not Satisfiable";

default:

case 500: return "Internal Server Error";

}

}

#ifdef CPPHTTPLIB\_ZLIB\_SUPPORT

inline bool can\_compress(const std::string &content\_type) {

return !content\_type.find("text/") || content\_type == "image/svg+xml" ||

content\_type == "application/javascript" ||

content\_type == "application/json" ||

content\_type == "application/xml" ||

content\_type == "application/xhtml+xml";

}

inline bool compress(std::string &content) {

z\_stream strm;

strm.zalloc = Z\_NULL;

strm.zfree = Z\_NULL;

strm.opaque = Z\_NULL;

auto ret = deflateInit2(&strm, Z\_DEFAULT\_COMPRESSION, Z\_DEFLATED, 31, 8,

Z\_DEFAULT\_STRATEGY);

if (ret != Z\_OK) { return false; }

strm.avail\_in = content.size();

strm.next\_in =

const\_cast<Bytef \*>(reinterpret\_cast<const Bytef \*>(content.data()));

std::string compressed;

std::array<char, 16384> buff{};

do {

strm.avail\_out = buff.size();

strm.next\_out = reinterpret\_cast<Bytef\*>(buff.data());

ret = deflate(&strm, Z\_FINISH);

assert(ret != Z\_STREAM\_ERROR);

compressed.append(buff.data(), buff.size() - strm.avail\_out);

} while (strm.avail\_out == 0);

assert(ret == Z\_STREAM\_END);

assert(strm.avail\_in == 0);

content.swap(compressed);

deflateEnd(&strm);

return true;

}

class decompressor {

public:

decompressor() {

strm.zalloc = Z\_NULL;

strm.zfree = Z\_NULL;

strm.opaque = Z\_NULL;

// 15 is the value of wbits, which should be at the maximum possible value

// to ensure that any gzip stream can be decoded. The offset of 16 specifies

// that the stream to decompress will be formatted with a gzip wrapper.

is\_valid\_ = inflateInit2(&strm, 16 + 15) == Z\_OK;

}

~decompressor() { inflateEnd(&strm); }

bool is\_valid() const { return is\_valid\_; }

template <typename T>

bool decompress(const char \*data, size\_t data\_length, T callback) {

int ret = Z\_OK;

strm.avail\_in = data\_length;

strm.next\_in = const\_cast<Bytef \*>(reinterpret\_cast<const Bytef \*>(data));

std::array<char, 16384> buff{};

do {

strm.avail\_out = buff.size();

strm.next\_out = reinterpret\_cast<Bytef\*>(buff.data());

ret = inflate(&strm, Z\_NO\_FLUSH);

assert(ret != Z\_STREAM\_ERROR);

switch (ret) {

case Z\_NEED\_DICT:

case Z\_DATA\_ERROR:

case Z\_MEM\_ERROR: inflateEnd(&strm); return false;

}

if (!callback(buff.data(), buff.size() - strm.avail\_out)) { return false; }

} while (strm.avail\_out == 0);

return ret == Z\_OK || ret == Z\_STREAM\_END;

}

private:

bool is\_valid\_;

z\_stream strm;

};

#endif

inline bool has\_header(const Headers &headers, const char \*key) {

return headers.find(key) != headers.end();

}

inline const char \*get\_header\_value(const Headers &headers, const char \*key,

size\_t id = 0, const char \*def = nullptr) {

auto it = headers.find(key);

std::advance(it, id);

if (it != headers.end()) { return it->second.c\_str(); }

return def;

}

inline uint64\_t get\_header\_value\_uint64(const Headers &headers, const char \*key,

int def = 0) {

auto it = headers.find(key);

if (it != headers.end()) {

return std::strtoull(it->second.data(), nullptr, 10);

}

return def;

}

inline bool read\_headers(Stream &strm, Headers &headers) {

static std::regex re(R"((.+?):\s\*(.+?)\s\*\r\n)");

const auto bufsiz = 2048;

char buf[bufsiz];

stream\_line\_reader line\_reader(strm, buf, bufsiz);

for (;;) {

if (!line\_reader.getline()) { return false; }

if (!strcmp(line\_reader.ptr(), "\r\n")) { break; }

std::cmatch m;

if (std::regex\_match(line\_reader.ptr(), m, re)) {

auto key = std::string(m[1]);

auto val = std::string(m[2]);

headers.emplace(key, val);

}

}

return true;

}

inline bool read\_content\_with\_length(Stream &strm, uint64\_t len,

Progress progress, ContentReceiver out) {

char buf[CPPHTTPLIB\_RECV\_BUFSIZ];

uint64\_t r = 0;

while (r < len) {

auto read\_len = static\_cast<size\_t>(len - r);

auto n = strm.read(buf, std::min(read\_len, CPPHTTPLIB\_RECV\_BUFSIZ));

if (n <= 0) { return false; }

if (!out(buf, n)) { return false; }

r += n;

if (progress) {

if (!progress(r, len)) { return false; }

}

}

return true;

}

inline void skip\_content\_with\_length(Stream &strm, uint64\_t len) {

char buf[CPPHTTPLIB\_RECV\_BUFSIZ];

uint64\_t r = 0;

while (r < len) {

auto read\_len = static\_cast<size\_t>(len - r);

auto n = strm.read(buf, std::min(read\_len, CPPHTTPLIB\_RECV\_BUFSIZ));

if (n <= 0) { return; }

r += n;

}

}

inline bool read\_content\_without\_length(Stream &strm, ContentReceiver out) {

char buf[CPPHTTPLIB\_RECV\_BUFSIZ];

for (;;) {

auto n = strm.read(buf, CPPHTTPLIB\_RECV\_BUFSIZ);

if (n < 0) {

return false;

} else if (n == 0) {

return true;

}

if (!out(buf, n)) { return false; }

}

return true;

}

inline bool read\_content\_chunked(Stream &strm, ContentReceiver out) {

const auto bufsiz = 16;

char buf[bufsiz];

stream\_line\_reader line\_reader(strm, buf, bufsiz);

if (!line\_reader.getline()) { return false; }

auto chunk\_len = std::stoi(line\_reader.ptr(), 0, 16);

while (chunk\_len > 0) {

if (!read\_content\_with\_length(strm, chunk\_len, nullptr, out)) {

return false;

}

if (!line\_reader.getline()) { return false; }

if (strcmp(line\_reader.ptr(), "\r\n")) { break; }

if (!line\_reader.getline()) { return false; }

chunk\_len = std::stoi(line\_reader.ptr(), 0, 16);

}

if (chunk\_len == 0) {

// Reader terminator after chunks

if (!line\_reader.getline() || strcmp(line\_reader.ptr(), "\r\n"))

return false;

}

return true;

}

inline bool is\_chunked\_transfer\_encoding(const Headers &headers) {

return !strcasecmp(get\_header\_value(headers, "Transfer-Encoding", 0, ""),

"chunked");

}

template <typename T>

bool read\_content(Stream &strm, T &x, size\_t payload\_max\_length, int &status,

Progress progress, ContentReceiver receiver) {

ContentReceiver out = [&](const char \*buf, size\_t n) {

return receiver(buf, n);

};

#ifdef CPPHTTPLIB\_ZLIB\_SUPPORT

detail::decompressor decompressor;

if (!decompressor.is\_valid()) {

status = 500;

return false;

}

if (x.get\_header\_value("Content-Encoding") == "gzip") {

out = [&](const char \*buf, size\_t n) {

return decompressor.decompress(

buf, n, [&](const char \*buf, size\_t n) { return receiver(buf, n); });

};

}

#else

if (x.get\_header\_value("Content-Encoding") == "gzip") {

status = 415;

return false;

}

#endif

auto ret = true;

auto exceed\_payload\_max\_length = false;

if (is\_chunked\_transfer\_encoding(x.headers)) {

ret = read\_content\_chunked(strm, out);

} else if (!has\_header(x.headers, "Content-Length")) {

ret = read\_content\_without\_length(strm, out);

} else {

auto len = get\_header\_value\_uint64(x.headers, "Content-Length", 0);

if (len > payload\_max\_length) {

exceed\_payload\_max\_length = true;

skip\_content\_with\_length(strm, len);

ret = false;

} else if (len > 0) {

ret = read\_content\_with\_length(strm, len, progress, out);

}

}

if (!ret) { status = exceed\_payload\_max\_length ? 413 : 400; }

return ret;

}

template <typename T>

inline int write\_headers(Stream &strm, const T &info, const Headers &headers) {

auto write\_len = 0;

for (const auto &x : info.headers) {

auto len =

strm.write\_format("%s: %s\r\n", x.first.c\_str(), x.second.c\_str());

if (len < 0) { return len; }

write\_len += len;

}

for (const auto &x : headers) {

auto len =

strm.write\_format("%s: %s\r\n", x.first.c\_str(), x.second.c\_str());

if (len < 0) { return len; }

write\_len += len;

}

auto len = strm.write("\r\n");

if (len < 0) { return len; }

write\_len += len;

return write\_len;

}

inline ssize\_t write\_content(Stream &strm,

ContentProviderWithCloser content\_provider,

size\_t offset, size\_t length) {

size\_t begin\_offset = offset;

size\_t end\_offset = offset + length;

while (offset < end\_offset) {

ssize\_t written\_length = 0;

content\_provider(

offset, end\_offset - offset,

[&](const char \*d, size\_t l) {

offset += l;

written\_length = strm.write(d, l);

},

[&](void) { written\_length = -1; });

if (written\_length < 0) { return written\_length; }

}

return static\_cast<ssize\_t>(offset - begin\_offset);

}

inline ssize\_t

write\_content\_chunked(Stream &strm,

ContentProviderWithCloser content\_provider) {

size\_t offset = 0;

auto data\_available = true;

ssize\_t total\_written\_length = 0;

while (data\_available) {

ssize\_t written\_length = 0;

content\_provider(

offset, 0,

[&](const char \*d, size\_t l) {

data\_available = l > 0;

offset += l;

// Emit chunked response header and footer for each chunk

auto chunk = from\_i\_to\_hex(l) + "\r\n" + std::string(d, l) + "\r\n";

written\_length = strm.write(chunk);

},

[&](void) {

data\_available = false;

written\_length = strm.write("0\r\n\r\n");

});

if (written\_length < 0) { return written\_length; }

total\_written\_length += written\_length;

}

return total\_written\_length;

}

template <typename T>

inline bool redirect(T &cli, const Request &req, Response &res,

const std::string &path) {

Request new\_req;

new\_req.method = req.method;

new\_req.path = path;

new\_req.headers = req.headers;

new\_req.body = req.body;

new\_req.redirect\_count = req.redirect\_count - 1;

new\_req.response\_handler = req.response\_handler;

new\_req.content\_receiver = req.content\_receiver;

new\_req.progress = req.progress;

Response new\_res;

auto ret = cli.send(new\_req, new\_res);

if (ret) { res = new\_res; }

return ret;

}

inline std::string encode\_url(const std::string &s) {

std::string result;

for (auto i = 0; s[i]; i++) {

switch (s[i]) {

case ' ': result += "%20"; break;

case '+': result += "%2B"; break;

case '\r': result += "%0D"; break;

case '\n': result += "%0A"; break;

case '\'': result += "%27"; break;

case ',': result += "%2C"; break;

case ':': result += "%3A"; break;

case ';': result += "%3B"; break;

default:

auto c = static\_cast<uint8\_t>(s[i]);

if (c >= 0x80) {

result += '%';

char hex[4];

size\_t len = snprintf(hex, sizeof(hex) - 1, "%02X", c);

assert(len == 2);

result.append(hex, len);

} else {

result += s[i];

}

break;

}

}

return result;

}

inline std::string decode\_url(const std::string &s) {

std::string result;

for (size\_t i = 0; i < s.size(); i++) {

if (s[i] == '%' && i + 1 < s.size()) {

if (s[i + 1] == 'u') {

int val = 0;

if (from\_hex\_to\_i(s, i + 2, 4, val)) {

// 4 digits Unicode codes

char buff[4];

size\_t len = to\_utf8(val, buff);

if (len > 0) { result.append(buff, len); }

i += 5; // 'u0000'

} else {

result += s[i];

}

} else {

int val = 0;

if (from\_hex\_to\_i(s, i + 1, 2, val)) {

// 2 digits hex codes

result += static\_cast<char>(val);

i += 2; // '00'

} else {

result += s[i];

}

}

} else if (s[i] == '+') {

result += ' ';

} else {

result += s[i];

}

}

return result;

}

inline void parse\_query\_text(const std::string &s, Params &params) {

split(&s[0], &s[s.size()], '&', [&](const char \*b, const char \*e) {

std::string key;

std::string val;

split(b, e, '=', [&](const char \*b, const char \*e) {

if (key.empty()) {

key.assign(b, e);

} else {

val.assign(b, e);

}

});

params.emplace(key, decode\_url(val));

});

}

inline bool parse\_multipart\_boundary(const std::string &content\_type,

std::string &boundary) {

auto pos = content\_type.find("boundary=");

if (pos == std::string::npos) { return false; }

boundary = content\_type.substr(pos + 9);

return true;

}

inline bool parse\_multipart\_formdata(const std::string &boundary,

const std::string &body,

MultipartFiles &files) {

static std::string dash = "--";

static std::string crlf = "\r\n";

static std::regex re\_content\_type("Content-Type: (.\*?)$",

std::regex\_constants::icase);

static std::regex re\_content\_disposition(

"Content-Disposition: form-data; name=\"(.\*?)\"(?:; filename=\"(.\*?)\")?",

std::regex\_constants::icase);

auto dash\_boundary = dash + boundary;

auto pos = body.find(dash\_boundary);

if (pos != 0) { return false; }

pos += dash\_boundary.size();

auto next\_pos = body.find(crlf, pos);

if (next\_pos == std::string::npos) { return false; }

pos = next\_pos + crlf.size();

while (pos < body.size()) {

next\_pos = body.find(crlf, pos);

if (next\_pos == std::string::npos) { return false; }

std::string name;

MultipartFile file;

auto header = body.substr(pos, (next\_pos - pos));

while (pos != next\_pos) {

std::smatch m;

if (std::regex\_match(header, m, re\_content\_type)) {

file.content\_type = m[1];

} else if (std::regex\_match(header, m, re\_content\_disposition)) {

name = m[1];

file.filename = m[2];

}

pos = next\_pos + crlf.size();

next\_pos = body.find(crlf, pos);

if (next\_pos == std::string::npos) { return false; }

header = body.substr(pos, (next\_pos - pos));

}

pos = next\_pos + crlf.size();

next\_pos = body.find(crlf + dash\_boundary, pos);

if (next\_pos == std::string::npos) { return false; }

file.offset = pos;

file.length = next\_pos - pos;

pos = next\_pos + crlf.size() + dash\_boundary.size();

next\_pos = body.find(crlf, pos);

if (next\_pos == std::string::npos) { return false; }

files.emplace(name, file);

pos = next\_pos + crlf.size();

}

return true;

}

inline bool parse\_range\_header(const std::string &s, Ranges &ranges) {

try {

static auto re\_first\_range =

std::regex(R"(bytes=(\d\*-\d\*(?:,\s\*\d\*-\d\*)\*))");

std::smatch m;

if (std::regex\_match(s, m, re\_first\_range)) {

auto pos = m.position(1);

auto len = m.length(1);

detail::split(

&s[pos], &s[pos + len], ',', [&](const char \*b, const char \*e) {

static auto re\_another\_range = std::regex(R"(\s\*(\d\*)-(\d\*))");

std::cmatch m;

if (std::regex\_match(b, e, m, re\_another\_range)) {

ssize\_t first = -1;

if (!m.str(1).empty()) {

first = static\_cast<ssize\_t>(std::stoll(m.str(1)));

}

ssize\_t last = -1;

if (!m.str(2).empty()) {

last = static\_cast<ssize\_t>(std::stoll(m.str(2)));

}

if (first != -1 && last != -1 && first > last) {

throw std::runtime\_error("invalid range error");

}

ranges.emplace\_back(std::make\_pair(first, last));

}

});

return true;

}

return false;

} catch (...) { return false; }

}

inline std::string to\_lower(const char \*beg, const char \*end) {

std::string out;

auto it = beg;

while (it != end) {

out += static\_cast<char>(::tolower(\*it));

it++;

}

return out;

}

inline std::string make\_multipart\_data\_boundary() {

static const char data[] =

"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";

std::random\_device seed\_gen;

std::mt19937 engine(seed\_gen());

std::string result = "--cpp-httplib-multipart-data-";

for (auto i = 0; i < 16; i++) {

result += data[engine() % (sizeof(data) - 1)];

}

return result;

}

inline std::pair<size\_t, size\_t>

get\_range\_offset\_and\_length(const Request &req, size\_t content\_length,

size\_t index) {

auto r = req.ranges[index];

if (r.first == -1 && r.second == -1) {

return std::make\_pair(0, content\_length);

}

if (r.first == -1) {

r.first = content\_length - r.second;

r.second = content\_length - 1;

}

if (r.second == -1) { r.second = content\_length - 1; }

return std::make\_pair(r.first, r.second - r.first + 1);

}

inline std::string make\_content\_range\_header\_field(size\_t offset, size\_t length,

size\_t content\_length) {

std::string field = "bytes ";

field += std::to\_string(offset);

field += "-";

field += std::to\_string(offset + length - 1);

field += "/";

field += std::to\_string(content\_length);

return field;

}

template <typename SToken, typename CToken, typename Content>

bool process\_multipart\_ranges\_data(const Request &req, Response &res,

const std::string &boundary,

const std::string &content\_type,

SToken stoken, CToken ctoken,

Content content) {

for (size\_t i = 0; i < req.ranges.size(); i++) {

ctoken("--");

stoken(boundary);

ctoken("\r\n");

if (!content\_type.empty()) {

ctoken("Content-Type: ");

stoken(content\_type);

ctoken("\r\n");

}

auto offsets = detail::get\_range\_offset\_and\_length(req, res.body.size(), i);

auto offset = offsets.first;

auto length = offsets.second;

ctoken("Content-Range: ");

stoken(make\_content\_range\_header\_field(offset, length, res.body.size()));

ctoken("\r\n");

ctoken("\r\n");

if (!content(offset, length)) { return false; }

ctoken("\r\n");

}

ctoken("--");

stoken(boundary);

ctoken("--\r\n");

return true;

}

inline std::string make\_multipart\_ranges\_data(const Request &req, Response &res,

const std::string &boundary,

const std::string &content\_type) {

std::string data;

process\_multipart\_ranges\_data(

req, res, boundary, content\_type,

[&](const std::string &token) { data += token; },

[&](const char \*token) { data += token; },

[&](size\_t offset, size\_t length) {

data += res.body.substr(offset, length);

return true;

});

return data;

}

inline size\_t

get\_multipart\_ranges\_data\_length(const Request &req, Response &res,

const std::string &boundary,

const std::string &content\_type) {

size\_t data\_length = 0;

process\_multipart\_ranges\_data(

req, res, boundary, content\_type,

[&](const std::string &token) { data\_length += token.size(); },

[&](const char \*token) { data\_length += strlen(token); },

[&](size\_t /\*offset\*/, size\_t length) {

data\_length += length;

return true;

});

return data\_length;

}

inline bool write\_multipart\_ranges\_data(Stream &strm, const Request &req,

Response &res,

const std::string &boundary,

const std::string &content\_type) {

return process\_multipart\_ranges\_data(

req, res, boundary, content\_type,

[&](const std::string &token) { strm.write(token); },

[&](const char \*token) { strm.write(token); },

[&](size\_t offset, size\_t length) {

return detail::write\_content(strm, res.content\_provider, offset,

length) >= 0;

});

}

inline std::pair<size\_t, size\_t>

get\_range\_offset\_and\_length(const Request &req, const Response &res,

size\_t index) {

auto r = req.ranges[index];

if (r.second == -1) { r.second = res.content\_length - 1; }

return std::make\_pair(r.first, r.second - r.first + 1);

}

#ifdef \_WIN32

class WSInit {

public:

WSInit() {

WSADATA wsaData;

WSAStartup(0x0002, &wsaData);

}

~WSInit() { WSACleanup(); }

};

static WSInit wsinit\_;

#endif

} // namespace detail

// Header utilities

inline std::pair<std::string, std::string> make\_range\_header(Ranges ranges) {

std::string field = "bytes=";

auto i = 0;

for (auto r : ranges) {

if (i != 0) { field += ", "; }

if (r.first != -1) { field += std::to\_string(r.first); }

field += '-';

if (r.second != -1) { field += std::to\_string(r.second); }

i++;

}

return std::make\_pair("Range", field);

}

inline std::pair<std::string, std::string>

make\_basic\_authentication\_header(const std::string &username,

const std::string &password) {

auto field = "Basic " + detail::base64\_encode(username + ":" + password);

return std::make\_pair("Authorization", field);

}

// Request implementation

inline bool Request::has\_header(const char \*key) const {

return detail::has\_header(headers, key);

}

inline std::string Request::get\_header\_value(const char \*key, size\_t id) const {

return detail::get\_header\_value(headers, key, id, "");

}

inline size\_t Request::get\_header\_value\_count(const char \*key) const {

auto r = headers.equal\_range(key);

return std::distance(r.first, r.second);

}

inline void Request::set\_header(const char \*key, const char \*val) {

headers.emplace(key, val);

}

inline void Request::set\_header(const char \*key, const std::string &val) {

headers.emplace(key, val);

}

inline bool Request::has\_param(const char \*key) const {

return params.find(key) != params.end();

}

inline std::string Request::get\_param\_value(const char \*key, size\_t id) const {

auto it = params.find(key);

std::advance(it, id);

if (it != params.end()) { return it->second; }

return std::string();

}

inline size\_t Request::get\_param\_value\_count(const char \*key) const {

auto r = params.equal\_range(key);

return std::distance(r.first, r.second);

}

inline bool Request::has\_file(const char \*key) const {

return files.find(key) != files.end();

}

inline MultipartFile Request::get\_file\_value(const char \*key) const {

auto it = files.find(key);

if (it != files.end()) { return it->second; }

return MultipartFile();

}

// Response implementation

inline bool Response::has\_header(const char \*key) const {

return headers.find(key) != headers.end();

}

inline std::string Response::get\_header\_value(const char \*key,

size\_t id) const {

return detail::get\_header\_value(headers, key, id, "");

}

inline size\_t Response::get\_header\_value\_count(const char \*key) const {

auto r = headers.equal\_range(key);

return std::distance(r.first, r.second);

}

inline void Response::set\_header(const char \*key, const char \*val) {

headers.emplace(key, val);

}

inline void Response::set\_header(const char \*key, const std::string &val) {

headers.emplace(key, val);

}

inline void Response::set\_redirect(const char \*url) {

set\_header("Location", url);

status = 302;

}

inline void Response::set\_content(const char \*s, size\_t n,

const char \*content\_type) {

body.assign(s, n);

set\_header("Content-Type", content\_type);

}

inline void Response::set\_content(const std::string &s,

const char \*content\_type) {

body = s;

set\_header("Content-Type", content\_type);

}

inline void Response::set\_content\_provider(

size\_t length,

std::function<void(size\_t offset, size\_t length, DataSink sink)> provider,

std::function<void()> resource\_releaser) {

assert(length > 0);

content\_length = length;

content\_provider = [provider](size\_t offset, size\_t length, DataSink sink,

Done) { provider(offset, length, sink); };

content\_provider\_resource\_releaser = resource\_releaser;

}

inline void Response::set\_chunked\_content\_provider(

std::function<void(size\_t offset, DataSink sink, Done done)> provider,

std::function<void()> resource\_releaser) {

content\_length = 0;

content\_provider = [provider](size\_t offset, size\_t, DataSink sink,

Done done) { provider(offset, sink, done); };

content\_provider\_resource\_releaser = resource\_releaser;

}

// Rstream implementation

template <typename... Args>

inline int Stream::write\_format(const char \*fmt, const Args &... args) {

std::array<char, 2048> buf;

#if defined(\_MSC\_VER) && \_MSC\_VER < 1900

auto n = \_snprintf\_s(buf, bufsiz, buf.size() - 1, fmt, args...);

#else

auto n = snprintf(buf.data(), buf.size() - 1, fmt, args...);

#endif

if (n <= 0) { return n; }

if (n >= static\_cast<int>(buf.size()) - 1) {

std::vector<char> glowable\_buf(buf.size());

while (n >= static\_cast<int>(glowable\_buf.size() - 1)) {

glowable\_buf.resize(glowable\_buf.size() \* 2);

#if defined(\_MSC\_VER) && \_MSC\_VER < 1900

n = \_snprintf\_s(&glowable\_buf[0], glowable\_buf.size(),

glowable\_buf.size() - 1, fmt, args...);

#else

n = snprintf(&glowable\_buf[0], glowable\_buf.size() - 1, fmt, args...);

#endif

}

return write(&glowable\_buf[0], n);

} else {

return write(buf.data(), n);

}

}

// Socket stream implementation

inline SocketStream::SocketStream(socket\_t sock, time\_t read\_timeout\_sec,

time\_t read\_timeout\_usec)

: sock\_(sock), read\_timeout\_sec\_(read\_timeout\_sec),

read\_timeout\_usec\_(read\_timeout\_usec) {}

inline SocketStream::~SocketStream() {}

inline int SocketStream::read(char \*ptr, size\_t size) {

if (detail::select\_read(sock\_, read\_timeout\_sec\_, read\_timeout\_usec\_) > 0) {

return recv(sock\_, ptr, static\_cast<int>(size), 0);

}

return -1;

}

inline int SocketStream::write(const char \*ptr, size\_t size) {

return send(sock\_, ptr, static\_cast<int>(size), 0);

}

inline int SocketStream::write(const char \*ptr) {

return write(ptr, strlen(ptr));

}

inline int SocketStream::write(const std::string &s) {

return write(s.data(), s.size());

}

inline std::string SocketStream::get\_remote\_addr() const {

return detail::get\_remote\_addr(sock\_);

}

// Buffer stream implementation

inline int BufferStream::read(char \*ptr, size\_t size) {

#if defined(\_MSC\_VER) && \_MSC\_VER < 1900

return static\_cast<int>(buffer.\_Copy\_s(ptr, size, size));

#else

return static\_cast<int>(buffer.copy(ptr, size));

#endif

}

inline int BufferStream::write(const char \*ptr, size\_t size) {

buffer.append(ptr, size);

return static\_cast<int>(size);

}

inline int BufferStream::write(const char \*ptr) {

return write(ptr, strlen(ptr));

}

inline int BufferStream::write(const std::string &s) {

return write(s.data(), s.size());

}

inline std::string BufferStream::get\_remote\_addr() const { return ""; }

inline const std::string &BufferStream::get\_buffer() const { return buffer; }

// HTTP server implementation

inline Server::Server()

: keep\_alive\_max\_count\_(CPPHTTPLIB\_KEEPALIVE\_MAX\_COUNT),

read\_timeout\_sec\_(CPPHTTPLIB\_READ\_TIMEOUT\_SECOND),

read\_timeout\_usec\_(CPPHTTPLIB\_READ\_TIMEOUT\_USECOND),

payload\_max\_length\_(CPPHTTPLIB\_PAYLOAD\_MAX\_LENGTH), is\_running\_(false),

svr\_sock\_(INVALID\_SOCKET) {

#ifndef \_WIN32

signal(SIGPIPE, SIG\_IGN);

#endif

new\_task\_queue = [] {

#if CPPHTTPLIB\_THREAD\_POOL\_COUNT > 0

return new ThreadPool(CPPHTTPLIB\_THREAD\_POOL\_COUNT);

#elif CPPHTTPLIB\_THREAD\_POOL\_COUNT == 0

return new Threads();

#else

return new NoThread();

#endif

};

}

inline Server::~Server() {}

inline Server &Server::Get(const char \*pattern, Handler handler) {

get\_handlers\_.push\_back(std::make\_pair(std::regex(pattern), handler));

return \*this;

}

inline Server &Server::Post(const char \*pattern, Handler handler) {

post\_handlers\_.push\_back(std::make\_pair(std::regex(pattern), handler));

return \*this;

}

inline Server &Server::Post(const char \*pattern,

HandlerWithContentReader handler) {

post\_handlers\_for\_content\_reader.push\_back(

std::make\_pair(std::regex(pattern), handler));

return \*this;

}

inline Server &Server::Put(const char \*pattern, Handler handler) {

put\_handlers\_.push\_back(std::make\_pair(std::regex(pattern), handler));

return \*this;

}

inline Server &Server::Put(const char \*pattern,

HandlerWithContentReader handler) {

put\_handlers\_for\_content\_reader.push\_back(

std::make\_pair(std::regex(pattern), handler));

return \*this;

}

inline Server &Server::Patch(const char \*pattern, Handler handler) {

patch\_handlers\_.push\_back(std::make\_pair(std::regex(pattern), handler));

return \*this;

}

inline Server &Server::Patch(const char \*pattern,

HandlerWithContentReader handler) {

patch\_handlers\_for\_content\_reader.push\_back(

std::make\_pair(std::regex(pattern), handler));

return \*this;

}

inline Server &Server::Delete(const char \*pattern, Handler handler) {

delete\_handlers\_.push\_back(std::make\_pair(std::regex(pattern), handler));

return \*this;

}

inline Server &Server::Options(const char \*pattern, Handler handler) {

options\_handlers\_.push\_back(std::make\_pair(std::regex(pattern), handler));

return \*this;

}

inline bool Server::set\_base\_dir(const char \*dir, const char \*mount\_point) {

if (detail::is\_dir(dir)) {

std::string mnt = mount\_point ? mount\_point : "/";

if (!mnt.empty() && mnt[0] == '/') {

base\_dirs\_.emplace\_back(mnt, dir);

return true;

}

}

return false;

}

inline void Server::set\_file\_request\_handler(Handler handler) {

file\_request\_handler\_ = std::move(handler);

}

inline void Server::set\_error\_handler(Handler handler) {

error\_handler\_ = std::move(handler);

}

inline void Server::set\_logger(Logger logger) { logger\_ = std::move(logger); }

inline void Server::set\_keep\_alive\_max\_count(size\_t count) {

keep\_alive\_max\_count\_ = count;

}

inline void Server::set\_read\_timeout(time\_t sec, time\_t usec) {

read\_timeout\_sec\_ = sec;

read\_timeout\_usec\_ = usec;

}

inline void Server::set\_payload\_max\_length(size\_t length) {

payload\_max\_length\_ = length;

}

inline bool Server::bind\_to\_port(const char \*host, int port, int socket\_flags) {

if (bind\_internal(host, port, socket\_flags) < 0) return false;

return true;

}

inline int Server::bind\_to\_any\_port(const char \*host, int socket\_flags) {

return bind\_internal(host, 0, socket\_flags);

}

inline bool Server::listen\_after\_bind() { return listen\_internal(); }

inline bool Server::listen(const char \*host, int port, int socket\_flags) {

return bind\_to\_port(host, port, socket\_flags) && listen\_internal();

}

inline bool Server::is\_running() const { return is\_running\_; }

inline void Server::stop() {

if (is\_running\_) {

assert(svr\_sock\_ != INVALID\_SOCKET);

std::atomic<socket\_t> sock(svr\_sock\_.exchange(INVALID\_SOCKET));

detail::shutdown\_socket(sock);

detail::close\_socket(sock);

}

}

inline bool Server::parse\_request\_line(const char \*s, Request &req) {

static std::regex re(

"(GET|HEAD|POST|PUT|DELETE|CONNECT|OPTIONS|TRACE|PATCH|PRI) "

"(([^?]+)(?:\\?(.\*?))?) (HTTP/1\\.[01])\r\n");

std::cmatch m;

if (std::regex\_match(s, m, re)) {

req.version = std::string(m[5]);

req.method = std::string(m[1]);

req.target = std::string(m[2]);

req.path = detail::decode\_url(m[3]);

// Parse query text

auto len = std::distance(m[4].first, m[4].second);

if (len > 0) { detail::parse\_query\_text(m[4], req.params); }

return true;

}

return false;

}

inline bool Server::write\_response(Stream &strm, bool last\_connection,

const Request &req, Response &res) {

assert(res.status != -1);

if (400 <= res.status && error\_handler\_) { error\_handler\_(req, res); }

// Response line

if (!strm.write\_format("HTTP/1.1 %d %s\r\n", res.status,

detail::status\_message(res.status))) {

return false;

}

// Headers

if (last\_connection || req.get\_header\_value("Connection") == "close") {

res.set\_header("Connection", "close");

}

if (!last\_connection && req.get\_header\_value("Connection") == "Keep-Alive") {

res.set\_header("Connection", "Keep-Alive");

}

if (!res.has\_header("Content-Type")) {

res.set\_header("Content-Type", "text/plain");

}

if (!res.has\_header("Accept-Ranges")) {

res.set\_header("Accept-Ranges", "bytes");

}

std::string content\_type;

std::string boundary;

if (req.ranges.size() > 1) {

boundary = detail::make\_multipart\_data\_boundary();

auto it = res.headers.find("Content-Type");

if (it != res.headers.end()) {

content\_type = it->second;

res.headers.erase(it);

}

res.headers.emplace("Content-Type",

"multipart/byteranges; boundary=" + boundary);

}

if (res.body.empty()) {

if (res.content\_length > 0) {

size\_t length = 0;

if (req.ranges.empty()) {

length = res.content\_length;

} else if (req.ranges.size() == 1) {

auto offsets =

detail::get\_range\_offset\_and\_length(req, res.content\_length, 0);

auto offset = offsets.first;

length = offsets.second;

auto content\_range = detail::make\_content\_range\_header\_field(

offset, length, res.content\_length);

res.set\_header("Content-Range", content\_range);

} else {

length = detail::get\_multipart\_ranges\_data\_length(req, res, boundary,

content\_type);

}

res.set\_header("Content-Length", std::to\_string(length));

} else {

if (res.content\_provider) {

res.set\_header("Transfer-Encoding", "chunked");

} else {

res.set\_header("Content-Length", "0");

}

}

} else {

if (req.ranges.empty()) {

;

} else if (req.ranges.size() == 1) {

auto offsets =

detail::get\_range\_offset\_and\_length(req, res.body.size(), 0);

auto offset = offsets.first;

auto length = offsets.second;

auto content\_range = detail::make\_content\_range\_header\_field(

offset, length, res.body.size());

res.set\_header("Content-Range", content\_range);

res.body = res.body.substr(offset, length);

} else {

res.body =

detail::make\_multipart\_ranges\_data(req, res, boundary, content\_type);

}

#ifdef CPPHTTPLIB\_ZLIB\_SUPPORT

// TODO: 'Accpet-Encoding' has gzip, not gzip;q=0

const auto &encodings = req.get\_header\_value("Accept-Encoding");

if (encodings.find("gzip") != std::string::npos &&

detail::can\_compress(res.get\_header\_value("Content-Type"))) {

if (detail::compress(res.body)) {

res.set\_header("Content-Encoding", "gzip");

}

}

#endif

auto length = std::to\_string(res.body.size());

res.set\_header("Content-Length", length);

}

if (!detail::write\_headers(strm, res, Headers())) { return false; }

// Body

if (req.method != "HEAD") {

if (!res.body.empty()) {

if (!strm.write(res.body)) { return false; }

} else if (res.content\_provider) {

if (!write\_content\_with\_provider(strm, req, res, boundary,

content\_type)) {

return false;

}

}

}

// Log

if (logger\_) { logger\_(req, res); }

return true;

}

inline bool

Server::write\_content\_with\_provider(Stream &strm, const Request &req,

Response &res, const std::string &boundary,

const std::string &content\_type) {

if (res.content\_length) {

if (req.ranges.empty()) {

if (detail::write\_content(strm, res.content\_provider, 0,

res.content\_length) < 0) {

return false;

}

} else if (req.ranges.size() == 1) {

auto offsets =

detail::get\_range\_offset\_and\_length(req, res.content\_length, 0);

auto offset = offsets.first;

auto length = offsets.second;

if (detail::write\_content(strm, res.content\_provider, offset, length) <

0) {

return false;

}

} else {

if (!detail::write\_multipart\_ranges\_data(strm, req, res, boundary,

content\_type)) {

return false;

}

}

} else {

if (detail::write\_content\_chunked(strm, res.content\_provider) < 0) {

return false;

}

}

return true;

}

inline bool Server::read\_content(Stream &strm, bool last\_connection,

Request &req, Response &res) {

if (!detail::read\_content(strm, req, payload\_max\_length\_, res.status,

Progress(), [&](const char \*buf, size\_t n) {

if (req.body.size() + n > req.body.max\_size()) {

return false;

}

req.body.append(buf, n);

return true;

})) {

return write\_response(strm, last\_connection, req, res);

}

const auto &content\_type = req.get\_header\_value("Content-Type");

if (!content\_type.find("application/x-www-form-urlencoded")) {

detail::parse\_query\_text(req.body, req.params);

} else if (!content\_type.find("multipart/form-data")) {

std::string boundary;

if (!detail::parse\_multipart\_boundary(content\_type, boundary) ||

!detail::parse\_multipart\_formdata(boundary, req.body, req.files)) {

res.status = 400;

return write\_response(strm, last\_connection, req, res);

}

}

return true;

}

inline bool

Server::read\_content\_with\_content\_receiver(Stream &strm, bool last\_connection,

Request &req, Response &res,

ContentReceiver receiver) {

if (!detail::read\_content(

strm, req, payload\_max\_length\_, res.status, Progress(),

[&](const char \*buf, size\_t n) { return receiver(buf, n); })) {

return write\_response(strm, last\_connection, req, res);

}

return true;

}

inline bool Server::handle\_file\_request(Request &req, Response &res) {

for (const auto& kv: base\_dirs\_) {

const auto& mount\_point = kv.first;

const auto& base\_dir = kv.second;

// Prefix match

if (!req.path.find(mount\_point)) {

std::string sub\_path = "/" + req.path.substr(mount\_point.size());

if (detail::is\_valid\_path(sub\_path)) {

auto path = base\_dir + sub\_path;

if (path.back() == '/') { path += "index.html"; }

if (detail::is\_file(path)) {

detail::read\_file(path, res.body);

auto type = detail::find\_content\_type(path);

if (type) { res.set\_header("Content-Type", type); }

res.status = 200;

if (file\_request\_handler\_) { file\_request\_handler\_(req, res); }

return true;

}

}

}

}

return false;

}

inline socket\_t Server::create\_server\_socket(const char \*host, int port,

int socket\_flags) const {

return detail::create\_socket(

host, port,

[](socket\_t sock, struct addrinfo &ai) -> bool {

if (::bind(sock, ai.ai\_addr, static\_cast<int>(ai.ai\_addrlen))) {

return false;

}

if (::listen(sock, 5)) { // Listen through 5 channels

return false;

}

return true;

},

socket\_flags);

}

inline int Server::bind\_internal(const char \*host, int port, int socket\_flags) {

if (!is\_valid()) { return -1; }

svr\_sock\_ = create\_server\_socket(host, port, socket\_flags);

if (svr\_sock\_ == INVALID\_SOCKET) { return -1; }

if (port == 0) {

struct sockaddr\_storage address;

socklen\_t len = sizeof(address);

if (getsockname(svr\_sock\_, reinterpret\_cast<struct sockaddr \*>(&address),

&len) == -1) {

return -1;

}

if (address.ss\_family == AF\_INET) {

return ntohs(reinterpret\_cast<struct sockaddr\_in \*>(&address)->sin\_port);

} else if (address.ss\_family == AF\_INET6) {

return ntohs(reinterpret\_cast<struct sockaddr\_in6 \*>(&address)->sin6\_port);

} else {

return -1;

}

} else {

return port;

}

}

inline bool Server::listen\_internal() {

auto ret = true;

is\_running\_ = true;

{

std::unique\_ptr<TaskQueue> task\_queue(new\_task\_queue());

for (;;) {

if (svr\_sock\_ == INVALID\_SOCKET) {

// The server socket was closed by 'stop' method.

break;

}

auto val = detail::select\_read(svr\_sock\_, 0, 100000);

if (val == 0) { // Timeout

continue;

}

socket\_t sock = accept(svr\_sock\_, nullptr, nullptr);

if (sock == INVALID\_SOCKET) {

if (errno == EMFILE) {

// The per-process limit of open file descriptors has been reached.

// Try to accept new connections after a short sleep.

std::this\_thread::sleep\_for(std::chrono::milliseconds(1));

continue;

}

if (svr\_sock\_ != INVALID\_SOCKET) {

detail::close\_socket(svr\_sock\_);

ret = false;

} else {

; // The server socket was closed by user.

}

break;

}

task\_queue->enqueue([=]() { process\_and\_close\_socket(sock); });

}

task\_queue->shutdown();

}

is\_running\_ = false;

return ret;

}

inline bool Server::routing(Request &req, Response &res, Stream &strm, bool last\_connection) {

// File handler

if (req.method == "GET" && handle\_file\_request(req, res)) { return true; }

// Content reader handler

if (req.method == "POST" || req.method == "PUT" || req.method == "PATCH") {

ContentReader content\_reader = [&](ContentReceiver receiver) {

return read\_content\_with\_content\_receiver(strm, last\_connection, req, res, receiver);

};

if (req.method == "POST") {

if (dispatch\_request\_for\_content\_reader(req, res, content\_reader,

post\_handlers\_for\_content\_reader)) {

return true;

}

} else if (req.method == "PUT") {

if (dispatch\_request\_for\_content\_reader(req, res, content\_reader,

put\_handlers\_for\_content\_reader)) {

return true;

}

} else if (req.method == "PATCH") {

if (dispatch\_request\_for\_content\_reader(

req, res, content\_reader, patch\_handlers\_for\_content\_reader)) {

return true;

}

}

}

// Read content into `req.body`

if (req.method == "POST" || req.method == "PUT" || req.method == "PATCH" || req.method == "PRI") {

if (!read\_content(strm, last\_connection, req, res)) {

return false;

}

}

// Regular handler

if (req.method == "GET" || req.method == "HEAD") {

return dispatch\_request(req, res, get\_handlers\_);

} else if (req.method == "POST") {

return dispatch\_request(req, res, post\_handlers\_);

} else if (req.method == "PUT") {

return dispatch\_request(req, res, put\_handlers\_);

} else if (req.method == "DELETE") {

return dispatch\_request(req, res, delete\_handlers\_);

} else if (req.method == "OPTIONS") {

return dispatch\_request(req, res, options\_handlers\_);

} else if (req.method == "PATCH") {

return dispatch\_request(req, res, patch\_handlers\_);

}

res.status = 400;

return false;

}

inline bool Server::dispatch\_request(Request &req, Response &res,

Handlers &handlers) {

for (const auto &x : handlers) {

const auto &pattern = x.first;

const auto &handler = x.second;

if (std::regex\_match(req.path, req.matches, pattern)) {

handler(req, res);

return true;

}

}

return false;

}

inline bool

Server::dispatch\_request\_for\_content\_reader(Request &req, Response &res,

ContentReader content\_reader,

HandersForContentReader &handlers) {

for (const auto &x : handlers) {

const auto &pattern = x.first;

const auto &handler = x.second;

if (std::regex\_match(req.path, req.matches, pattern)) {

handler(req, res, content\_reader);

return true;

}

}

return false;

}

inline bool

Server::process\_request(Stream &strm, bool last\_connection,

bool &connection\_close,

const std::function<void(Request &)>& setup\_request) {

std::array<char, 2048> buf{};

detail::stream\_line\_reader line\_reader(strm, buf.data(), buf.size());

// Connection has been closed on client

if (!line\_reader.getline()) { return false; }

Request req;

Response res;

res.version = "HTTP/1.1";

// Check if the request URI doesn't exceed the limit

if (line\_reader.size() > CPPHTTPLIB\_REQUEST\_URI\_MAX\_LENGTH) {

Headers dummy;

detail::read\_headers(strm, dummy);

res.status = 414;

return write\_response(strm, last\_connection, req, res);

}

// Request line and headers

if (!parse\_request\_line(line\_reader.ptr(), req) ||

!detail::read\_headers(strm, req.headers)) {

res.status = 400;

return write\_response(strm, last\_connection, req, res);

}

if (req.get\_header\_value("Connection") == "close") {

connection\_close = true;

}

if (req.version == "HTTP/1.0" &&

req.get\_header\_value("Connection") != "Keep-Alive") {

connection\_close = true;

}

req.set\_header("REMOTE\_ADDR", strm.get\_remote\_addr());

if (req.has\_header("Range")) {

const auto &range\_header\_value = req.get\_header\_value("Range");

if (!detail::parse\_range\_header(range\_header\_value, req.ranges)) {

// TODO: error

}

}

if (setup\_request) { setup\_request(req); }

// Rounting

if (routing(req, res, strm, last\_connection)) {

if (res.status == -1) { res.status = req.ranges.empty() ? 200 : 206; }

} else {

if (res.status == -1) { res.status = 404; }

}

return write\_response(strm, last\_connection, req, res);

}

inline bool Server::is\_valid() const { return true; }

inline bool Server::process\_and\_close\_socket(socket\_t sock) {

return detail::process\_and\_close\_socket(

false, sock, keep\_alive\_max\_count\_, read\_timeout\_sec\_, read\_timeout\_usec\_,

[this](Stream &strm, bool last\_connection, bool &connection\_close) {

return process\_request(strm, last\_connection, connection\_close,

nullptr);

});

}

// HTTP client implementation

inline Client::Client(const char \*host, int port, time\_t timeout\_sec)

: host\_(host), port\_(port), timeout\_sec\_(timeout\_sec),

host\_and\_port\_(host\_ + ":" + std::to\_string(port\_)),

keep\_alive\_max\_count\_(CPPHTTPLIB\_KEEPALIVE\_MAX\_COUNT),

read\_timeout\_sec\_(CPPHTTPLIB\_READ\_TIMEOUT\_SECOND),

read\_timeout\_usec\_(CPPHTTPLIB\_READ\_TIMEOUT\_USECOND),

follow\_location\_(false) {}

inline Client::~Client() {}

inline bool Client::is\_valid() const { return true; }

inline socket\_t Client::create\_client\_socket() const {

return detail::create\_socket(

host\_.c\_str(), port\_, [=](socket\_t sock, struct addrinfo &ai) -> bool {

detail::set\_nonblocking(sock, true);

auto ret = connect(sock, ai.ai\_addr, static\_cast<int>(ai.ai\_addrlen));

if (ret < 0) {

if (detail::is\_connection\_error() ||

!detail::wait\_until\_socket\_is\_ready(sock, timeout\_sec\_, 0)) {

detail::close\_socket(sock);

return false;

}

}

detail::set\_nonblocking(sock, false);

return true;

});

}

inline bool Client::read\_response\_line(Stream &strm, Response &res) {

std::array<char, 2048> buf;

detail::stream\_line\_reader line\_reader(strm, buf.data(), buf.size());

if (!line\_reader.getline()) { return false; }

const static std::regex re("(HTTP/1\\.[01]) (\\d+?) .\*\r\n");

std::cmatch m;

if (std::regex\_match(line\_reader.ptr(), m, re)) {

res.version = std::string(m[1]);

res.status = std::stoi(std::string(m[2]));

}

return true;

}

inline bool Client::send(const Request &req, Response &res) {

if (req.path.empty()) { return false; }

auto sock = create\_client\_socket();

if (sock == INVALID\_SOCKET) { return false; }

auto ret = process\_and\_close\_socket(

sock, 1, [&](Stream &strm, bool last\_connection, bool &connection\_close) {

return process\_request(strm, req, res, last\_connection,

connection\_close);

});

if (ret && follow\_location\_ && (300 < res.status && res.status < 400)) {

ret = redirect(req, res);

}

return ret;

}

inline bool Client::send(const std::vector<Request> &requests,

std::vector<Response> &responses) {

size\_t i = 0;

while (i < requests.size()) {

auto sock = create\_client\_socket();

if (sock == INVALID\_SOCKET) { return false; }

if (!process\_and\_close\_socket(

sock, requests.size() - i,

[&](Stream &strm, bool last\_connection,

bool &connection\_close) -> bool {

auto &req = requests[i];

auto res = Response();

i++;

if (req.path.empty()) { return false; }

auto ret = process\_request(strm, req, res, last\_connection,

connection\_close);

if (ret && follow\_location\_ &&

(300 < res.status && res.status < 400)) {

ret = redirect(req, res);

}

if (ret) { responses.emplace\_back(std::move(res)); }

return ret;

})) {

return false;

}

}

return true;

}

inline bool Client::redirect(const Request &req, Response &res) {

if (req.redirect\_count == 0) { return false; }

auto location = res.get\_header\_value("location");

if (location.empty()) { return false; }

std::regex re(

R"(^(?:([^:/?#]+):)?(?://([^/?#]\*))?([^?#]\*(?:\?[^#]\*)?)(?:#.\*)?)");

auto scheme = is\_ssl() ? "https" : "http";

std::smatch m;

if (regex\_match(location, m, re)) {

auto next\_scheme = m[1].str();

auto next\_host = m[2].str();

auto next\_path = m[3].str();

if (next\_host.empty()) { next\_host = host\_; }

if (next\_path.empty()) { next\_path = "/"; }

if (next\_scheme == scheme && next\_host == host\_) {

return detail::redirect(\*this, req, res, next\_path);

} else {

if (next\_scheme == "https") {

#ifdef CPPHTTPLIB\_OPENSSL\_SUPPORT

SSLClient cli(next\_host.c\_str());

cli.follow\_location(true);

return detail::redirect(cli, req, res, next\_path);

#else

return false;

#endif

} else {

Client cli(next\_host.c\_str());

cli.follow\_location(true);

return detail::redirect(cli, req, res, next\_path);

}

}

}

return false;

}

inline void Client::write\_request(Stream &strm, const Request &req,

bool last\_connection) {

BufferStream bstrm;

// Request line

auto path = detail::encode\_url(req.path);

bstrm.write\_format("%s %s HTTP/1.1\r\n", req.method.c\_str(), path.c\_str());

// Additonal headers

Headers headers;

if (last\_connection) { headers.emplace("Connection", "close"); }

if (!req.has\_header("Host")) {

if (is\_ssl()) {

if (port\_ == 443) {

headers.emplace("Host", host\_);

} else {

headers.emplace("Host", host\_and\_port\_);

}

} else {

if (port\_ == 80) {

headers.emplace("Host", host\_);

} else {

headers.emplace("Host", host\_and\_port\_);

}

}

}

if (!req.has\_header("Accept")) { headers.emplace("Accept", "\*/\*"); }

if (!req.has\_header("User-Agent")) {

headers.emplace("User-Agent", "cpp-httplib/0.2");

}

if (req.body.empty()) {

if (req.content\_provider) {

auto length = std::to\_string(req.content\_length);

headers.emplace("Content-Length", length);

} else {

headers.emplace("Content-Length", "0");

}

} else {

if (!req.has\_header("Content-Type")) {

headers.emplace("Content-Type", "text/plain");

}

if (!req.has\_header("Content-Length")) {

auto length = std::to\_string(req.body.size());

headers.emplace("Content-Length", length);

}

}

detail::write\_headers(bstrm, req, headers);

// Flush buffer

auto &data = bstrm.get\_buffer();

strm.write(data.data(), data.size());

// Body

if (req.body.empty()) {

if (req.content\_provider) {

size\_t offset = 0;

size\_t end\_offset = req.content\_length;

while (offset < end\_offset) {

req.content\_provider(offset, end\_offset - offset,

[&](const char \*d, size\_t l) {

auto written\_length = strm.write(d, l);

offset += written\_length;

});

}

}

} else {

strm.write(req.body);

}

}

inline std::shared\_ptr<Response> Client::send\_with\_content\_provider(

const char \*method, const char \*path, const Headers &headers,

const std::string &body, size\_t content\_length,

ContentProvider content\_provider, const char \*content\_type, bool compress) {

#ifndef CPPHTTPLIB\_ZLIB\_SUPPORT

(void)compress;

#endif

Request req;

req.method = method;

req.headers = headers;

req.path = path;

req.headers.emplace("Content-Type", content\_type);

#ifdef CPPHTTPLIB\_ZLIB\_SUPPORT

if (compress) {

if (content\_provider) {

size\_t offset = 0;

while (offset < content\_length) {

content\_provider(offset, content\_length - offset,

[&](const char \*data, size\_t data\_len) {

req.body.append(data, data\_len);

offset += data\_len;

});

}

} else {

req.body = body;

}

if (!detail::compress(req.body)) { return nullptr; }

req.headers.emplace("Content-Encoding", "gzip");

} else

#endif

{

if (content\_provider) {

req.content\_length = content\_length;

req.content\_provider = content\_provider;

} else {

req.body = body;

}

}

auto res = std::make\_shared<Response>();

return send(req, \*res) ? res : nullptr;

}

inline bool Client::process\_request(Stream &strm, const Request &req,

Response &res, bool last\_connection,

bool &connection\_close) {

// Send request

write\_request(strm, req, last\_connection);

// Receive response and headers

if (!read\_response\_line(strm, res) ||

!detail::read\_headers(strm, res.headers)) {

return false;

}

if (res.get\_header\_value("Connection") == "close" ||

res.version == "HTTP/1.0") {

connection\_close = true;

}

if (req.response\_handler) {

if (!req.response\_handler(res)) { return false; }

}

// Body

if (req.method != "HEAD") {

ContentReceiver out = [&](const char \*buf, size\_t n) {

if (res.body.size() + n > res.body.max\_size()) { return false; }

res.body.append(buf, n);

return true;

};

if (req.content\_receiver) {

out = [&](const char \*buf, size\_t n) {

return req.content\_receiver(buf, n);

};

}

int dummy\_status;

if (!detail::read\_content(strm, res, std::numeric\_limits<size\_t>::max(),

dummy\_status, req.progress, out)) {

return false;

}

}

return true;

}

inline bool Client::process\_and\_close\_socket(

socket\_t sock, size\_t request\_count,

std::function<bool(Stream &strm, bool last\_connection,

bool &connection\_close)>

callback) {

request\_count = std::min(request\_count, keep\_alive\_max\_count\_);

return detail::process\_and\_close\_socket(true, sock, request\_count,

read\_timeout\_sec\_, read\_timeout\_usec\_,

callback);

}

inline bool Client::is\_ssl() const { return false; }

inline std::shared\_ptr<Response> Client::Get(const char \*path) {

Progress dummy;

return Get(path, Headers(), dummy);

}

inline std::shared\_ptr<Response> Client::Get(const char \*path,

Progress progress) {

return Get(path, Headers(), std::move(progress));

}

inline std::shared\_ptr<Response> Client::Get(const char \*path,

const Headers &headers) {

Progress dummy;

return Get(path, headers, dummy);

}

inline std::shared\_ptr<Response>

Client::Get(const char \*path, const Headers &headers, Progress progress) {

Request req;

req.method = "GET";

req.path = path;

req.headers = headers;

req.progress = std::move(progress);

auto res = std::make\_shared<Response>();

return send(req, \*res) ? res : nullptr;

}

inline std::shared\_ptr<Response> Client::Get(const char \*path,

ContentReceiver content\_receiver) {

Progress dummy;

return Get(path, Headers(), nullptr, std::move(content\_receiver), dummy);

}

inline std::shared\_ptr<Response> Client::Get(const char \*path,

ContentReceiver content\_receiver,

Progress progress) {

return Get(path, Headers(), nullptr, std::move(content\_receiver), progress);

}

inline std::shared\_ptr<Response> Client::Get(const char \*path,

const Headers &headers,

ContentReceiver content\_receiver) {

Progress dummy;

return Get(path, headers, nullptr, std::move(content\_receiver), dummy);

}

inline std::shared\_ptr<Response> Client::Get(const char \*path,

const Headers &headers,

ContentReceiver content\_receiver,

Progress progress) {

return Get(path, headers, nullptr, std::move(content\_receiver), progress);

}

inline std::shared\_ptr<Response> Client::Get(const char \*path,

const Headers &headers,

ResponseHandler response\_handler,

ContentReceiver content\_receiver) {

Progress dummy;

return Get(path, headers, std::move(response\_handler), content\_receiver, dummy);

}

inline std::shared\_ptr<Response> Client::Get(const char \*path,

const Headers &headers,

ResponseHandler response\_handler,

ContentReceiver content\_receiver,

Progress progress) {

Request req;

req.method = "GET";

req.path = path;

req.headers = headers;

req.response\_handler = std::move(response\_handler);

req.content\_receiver = std::move(content\_receiver);

req.progress = std::move(progress);

auto res = std::make\_shared<Response>();

return send(req, \*res) ? res : nullptr;

}

inline std::shared\_ptr<Response> Client::Head(const char \*path) {

return Head(path, Headers());

}

inline std::shared\_ptr<Response> Client::Head(const char \*path,

const Headers &headers) {

Request req;

req.method = "HEAD";

req.headers = headers;

req.path = path;

auto res = std::make\_shared<Response>();

return send(req, \*res) ? res : nullptr;

}

inline std::shared\_ptr<Response> Client::Post(const char \*path,

const std::string &body,

const char \*content\_type,

bool compress) {

return Post(path, Headers(), body, content\_type, compress);

}

inline std::shared\_ptr<Response>

Client::Post(const char \*path, const Headers &headers, const std::string &body,

const char \*content\_type, bool compress) {

return send\_with\_content\_provider("POST", path, headers, body, 0, nullptr,

content\_type, compress);

}

inline std::shared\_ptr<Response>

Client::Post(const char \*path, const Params &params, bool compress) {

return Post(path, Headers(), params, compress);

}

inline std::shared\_ptr<Response> Client::Post(const char \*path,

size\_t content\_length,

ContentProvider content\_provider,

const char \*content\_type,

bool compress) {

return Post(path, Headers(), content\_length, content\_provider, content\_type,

compress);

}

inline std::shared\_ptr<Response>

Client::Post(const char \*path, const Headers &headers, size\_t content\_length,

ContentProvider content\_provider, const char \*content\_type,

bool compress) {

return send\_with\_content\_provider("POST", path, headers, std::string(),

content\_length, content\_provider,

content\_type, compress);

}

inline std::shared\_ptr<Response> Client::Post(const char \*path,

const Headers &headers,

const Params &params,

bool compress) {

std::string query;

for (auto it = params.begin(); it != params.end(); ++it) {

if (it != params.begin()) { query += "&"; }

query += it->first;

query += "=";

query += detail::encode\_url(it->second);

}

return Post(path, headers, query, "application/x-www-form-urlencoded",

compress);

}

inline std::shared\_ptr<Response>

Client::Post(const char \*path, const MultipartFormDataItems &items,

bool compress) {

return Post(path, Headers(), items, compress);

}

inline std::shared\_ptr<Response>

Client::Post(const char \*path, const Headers &headers,

const MultipartFormDataItems &items, bool compress) {

auto boundary = detail::make\_multipart\_data\_boundary();

std::string body;

for (const auto &item : items) {

body += "--" + boundary + "\r\n";

body += "Content-Disposition: form-data; name=\"" + item.name + "\"";

if (!item.filename.empty()) {

body += "; filename=\"" + item.filename + "\"";

}

body += "\r\n";

if (!item.content\_type.empty()) {

body += "Content-Type: " + item.content\_type + "\r\n";

}

body += "\r\n";

body += item.content + "\r\n";

}

body += "--" + boundary + "--\r\n";

std::string content\_type = "multipart/form-data; boundary=" + boundary;

return Post(path, headers, body, content\_type.c\_str(), compress);

}

inline std::shared\_ptr<Response> Client::Put(const char \*path,

const std::string &body,

const char \*content\_type,

bool compress) {

return Put(path, Headers(), body, content\_type, compress);

}

inline std::shared\_ptr<Response>

Client::Put(const char \*path, const Headers &headers, const std::string &body,

const char \*content\_type, bool compress) {

return send\_with\_content\_provider("PUT", path, headers, body, 0, nullptr,

content\_type, compress);

}

inline std::shared\_ptr<Response> Client::Put(const char \*path,

size\_t content\_length,

ContentProvider content\_provider,

const char \*content\_type,

bool compress) {

return Put(path, Headers(), content\_length, content\_provider, content\_type,

compress);

}

inline std::shared\_ptr<Response>

Client::Put(const char \*path, const Headers &headers, size\_t content\_length,

ContentProvider content\_provider, const char \*content\_type,

bool compress) {

return send\_with\_content\_provider("PUT", path, headers, std::string(),

content\_length, content\_provider,

content\_type, compress);

}

inline std::shared\_ptr<Response> Client::Patch(const char \*path,

const std::string &body,

const char \*content\_type,

bool compress) {

return Patch(path, Headers(), body, content\_type, compress);

}

inline std::shared\_ptr<Response>

Client::Patch(const char \*path, const Headers &headers, const std::string &body,

const char \*content\_type, bool compress) {

return send\_with\_content\_provider("PATCH", path, headers, body, 0, nullptr,

content\_type, compress);

}

inline std::shared\_ptr<Response> Client::Patch(const char \*path,

size\_t content\_length,

ContentProvider content\_provider,

const char \*content\_type,

bool compress) {

return Patch(path, Headers(), content\_length, content\_provider, content\_type,

compress);

}

inline std::shared\_ptr<Response>

Client::Patch(const char \*path, const Headers &headers, size\_t content\_length,

ContentProvider content\_provider, const char \*content\_type,

bool compress) {

return send\_with\_content\_provider("PATCH", path, headers, std::string(),

content\_length, content\_provider,

content\_type, compress);

}

inline std::shared\_ptr<Response> Client::Delete(const char \*path) {

return Delete(path, Headers(), std::string(), nullptr);

}

inline std::shared\_ptr<Response> Client::Delete(const char \*path,

const std::string &body,

const char \*content\_type) {

return Delete(path, Headers(), body, content\_type);

}

inline std::shared\_ptr<Response> Client::Delete(const char \*path,

const Headers &headers) {

return Delete(path, headers, std::string(), nullptr);

}

inline std::shared\_ptr<Response> Client::Delete(const char \*path,

const Headers &headers,

const std::string &body,

const char \*content\_type) {

Request req;

req.method = "DELETE";

req.headers = headers;

req.path = path;

if (content\_type) { req.headers.emplace("Content-Type", content\_type); }

req.body = body;

auto res = std::make\_shared<Response>();

return send(req, \*res) ? res : nullptr;

}

inline std::shared\_ptr<Response> Client::Options(const char \*path) {

return Options(path, Headers());

}

inline std::shared\_ptr<Response> Client::Options(const char \*path,

const Headers &headers) {

Request req;

req.method = "OPTIONS";

req.path = path;

req.headers = headers;

auto res = std::make\_shared<Response>();

return send(req, \*res) ? res : nullptr;

}

inline void Client::set\_keep\_alive\_max\_count(size\_t count) {

keep\_alive\_max\_count\_ = count;

}

inline void Client::set\_read\_timeout(time\_t sec, time\_t usec) {

read\_timeout\_sec\_ = sec;

read\_timeout\_usec\_ = usec;

}

inline void Client::follow\_location(bool on) { follow\_location\_ = on; }

/\*

\* SSL Implementation

\*/

#ifdef CPPHTTPLIB\_OPENSSL\_SUPPORT

namespace detail {

template <typename U, typename V, typename T>

inline bool process\_and\_close\_socket\_ssl(

bool is\_client\_request, socket\_t sock, size\_t keep\_alive\_max\_count,

time\_t read\_timeout\_sec, time\_t read\_timeout\_usec, SSL\_CTX \*ctx,

std::mutex &ctx\_mutex, U SSL\_connect\_or\_accept, V setup, T callback) {

assert(keep\_alive\_max\_count > 0);

SSL \*ssl = nullptr;

{

std::lock\_guard<std::mutex> guard(ctx\_mutex);

ssl = SSL\_new(ctx);

}

if (!ssl) {

close\_socket(sock);

return false;

}

auto bio = BIO\_new\_socket(static\_cast<int>(sock), BIO\_NOCLOSE);

SSL\_set\_bio(ssl, bio, bio);

if (!setup(ssl)) {

SSL\_shutdown(ssl);

{

std::lock\_guard<std::mutex> guard(ctx\_mutex);

SSL\_free(ssl);

}

close\_socket(sock);

return false;

}

bool ret = false;

if (SSL\_connect\_or\_accept(ssl) == 1) {

if (keep\_alive\_max\_count > 1) {

auto count = keep\_alive\_max\_count;

while (count > 0 &&

(is\_client\_request ||

detail::select\_read(sock, CPPHTTPLIB\_KEEPALIVE\_TIMEOUT\_SECOND,

CPPHTTPLIB\_KEEPALIVE\_TIMEOUT\_USECOND) > 0)) {

SSLSocketStream strm(sock, ssl, read\_timeout\_sec, read\_timeout\_usec);

auto last\_connection = count == 1;

auto connection\_close = false;

ret = callback(ssl, strm, last\_connection, connection\_close);

if (!ret || connection\_close) { break; }

count--;

}

} else {

SSLSocketStream strm(sock, ssl, read\_timeout\_sec, read\_timeout\_usec);

auto dummy\_connection\_close = false;

ret = callback(ssl, strm, true, dummy\_connection\_close);

}

}

SSL\_shutdown(ssl);

{

std::lock\_guard<std::mutex> guard(ctx\_mutex);

SSL\_free(ssl);

}

close\_socket(sock);

return ret;

}

#if OPENSSL\_VERSION\_NUMBER < 0x10100000L

static std::shared\_ptr<std::vector<std::mutex>> openSSL\_locks\_;

class SSLThreadLocks {

public:

SSLThreadLocks() {

openSSL\_locks\_ =

std::make\_shared<std::vector<std::mutex>>(CRYPTO\_num\_locks());

CRYPTO\_set\_locking\_callback(locking\_callback);

}

~SSLThreadLocks() { CRYPTO\_set\_locking\_callback(nullptr); }

private:

static void locking\_callback(int mode, int type, const char \* /\*file\*/,

int /\*line\*/) {

auto &locks = \*openSSL\_locks\_;

if (mode & CRYPTO\_LOCK) {

locks[type].lock();

} else {

locks[type].unlock();

}

}

};

#endif

class SSLInit {

public:

SSLInit() {

#if OPENSSL\_VERSION\_NUMBER < 0x1010001fL

SSL\_load\_error\_strings();

SSL\_library\_init();

#else

OPENSSL\_init\_ssl(

OPENSSL\_INIT\_LOAD\_SSL\_STRINGS | OPENSSL\_INIT\_LOAD\_CRYPTO\_STRINGS, NULL);

#endif

}

~SSLInit() {

#if OPENSSL\_VERSION\_NUMBER < 0x1010001fL

ERR\_free\_strings();

#endif

}

private:

#if OPENSSL\_VERSION\_NUMBER < 0x10100000L

SSLThreadLocks thread\_init\_;

#endif

};

static SSLInit sslinit\_;

} // namespace detail

// SSL socket stream implementation

inline SSLSocketStream::SSLSocketStream(socket\_t sock, SSL \*ssl,

time\_t read\_timeout\_sec,

time\_t read\_timeout\_usec)

: sock\_(sock), ssl\_(ssl), read\_timeout\_sec\_(read\_timeout\_sec),

read\_timeout\_usec\_(read\_timeout\_usec) {}

inline SSLSocketStream::~SSLSocketStream() {}

inline int SSLSocketStream::read(char \*ptr, size\_t size) {

if (SSL\_pending(ssl\_) > 0 ||

detail::select\_read(sock\_, read\_timeout\_sec\_, read\_timeout\_usec\_) > 0) {

return SSL\_read(ssl\_, ptr, static\_cast<int>(size));

}

return -1;

}

inline int SSLSocketStream::write(const char \*ptr, size\_t size) {

return SSL\_write(ssl\_, ptr, static\_cast<int>(size));

}

inline int SSLSocketStream::write(const char \*ptr) {

return write(ptr, strlen(ptr));

}

inline int SSLSocketStream::write(const std::string &s) {

return write(s.data(), s.size());

}

inline std::string SSLSocketStream::get\_remote\_addr() const {

return detail::get\_remote\_addr(sock\_);

}

// SSL HTTP server implementation

inline SSLServer::SSLServer(const char \*cert\_path, const char \*private\_key\_path,

const char \*client\_ca\_cert\_file\_path,

const char \*client\_ca\_cert\_dir\_path) {

ctx\_ = SSL\_CTX\_new(SSLv23\_server\_method());

if (ctx\_) {

SSL\_CTX\_set\_options(ctx\_,

SSL\_OP\_ALL | SSL\_OP\_NO\_SSLv2 | SSL\_OP\_NO\_SSLv3 |

SSL\_OP\_NO\_COMPRESSION |

SSL\_OP\_NO\_SESSION\_RESUMPTION\_ON\_RENEGOTIATION);

// auto ecdh = EC\_KEY\_new\_by\_curve\_name(NID\_X9\_62\_prime256v1);

// SSL\_CTX\_set\_tmp\_ecdh(ctx\_, ecdh);

// EC\_KEY\_free(ecdh);

if (SSL\_CTX\_use\_certificate\_chain\_file(ctx\_, cert\_path) != 1 ||

SSL\_CTX\_use\_PrivateKey\_file(ctx\_, private\_key\_path, SSL\_FILETYPE\_PEM) !=

1) {

SSL\_CTX\_free(ctx\_);

ctx\_ = nullptr;

} else if (client\_ca\_cert\_file\_path || client\_ca\_cert\_dir\_path) {

// if (client\_ca\_cert\_file\_path) {

// auto list = SSL\_load\_client\_CA\_file(client\_ca\_cert\_file\_path);

// SSL\_CTX\_set\_client\_CA\_list(ctx\_, list);

// }

SSL\_CTX\_load\_verify\_locations(ctx\_, client\_ca\_cert\_file\_path,

client\_ca\_cert\_dir\_path);

SSL\_CTX\_set\_verify(

ctx\_,

SSL\_VERIFY\_PEER |

SSL\_VERIFY\_FAIL\_IF\_NO\_PEER\_CERT, // SSL\_VERIFY\_CLIENT\_ONCE,

nullptr);

}

}

}

inline SSLServer::~SSLServer() {

if (ctx\_) { SSL\_CTX\_free(ctx\_); }

}

inline bool SSLServer::is\_valid() const { return ctx\_; }

inline bool SSLServer::process\_and\_close\_socket(socket\_t sock) {

return detail::process\_and\_close\_socket\_ssl(

false, sock, keep\_alive\_max\_count\_, read\_timeout\_sec\_, read\_timeout\_usec\_,

ctx\_, ctx\_mutex\_, SSL\_accept, [](SSL \* /\*ssl\*/) { return true; },

[this](SSL \*ssl, Stream &strm, bool last\_connection,

bool &connection\_close) {

return process\_request(strm, last\_connection, connection\_close,

[&](Request &req) { req.ssl = ssl; });

});

}

// SSL HTTP client implementation

inline SSLClient::SSLClient(const char \*host, int port, time\_t timeout\_sec,

const char \*client\_cert\_path,

const char \*client\_key\_path)

: Client(host, port, timeout\_sec) {

ctx\_ = SSL\_CTX\_new(SSLv23\_client\_method());

detail::split(&host\_[0], &host\_[host\_.size()], '.',

[&](const char \*b, const char \*e) {

host\_components\_.emplace\_back(std::string(b, e));

});

if (client\_cert\_path && client\_key\_path) {

if (SSL\_CTX\_use\_certificate\_file(ctx\_, client\_cert\_path,

SSL\_FILETYPE\_PEM) != 1 ||

SSL\_CTX\_use\_PrivateKey\_file(ctx\_, client\_key\_path, SSL\_FILETYPE\_PEM) !=

1) {

SSL\_CTX\_free(ctx\_);

ctx\_ = nullptr;

}

}

}

inline SSLClient::~SSLClient() {

if (ctx\_) { SSL\_CTX\_free(ctx\_); }

}

inline bool SSLClient::is\_valid() const { return ctx\_; }

inline void SSLClient::set\_ca\_cert\_path(const char \*ca\_cert\_file\_path,

const char \*ca\_cert\_dir\_path) {

if (ca\_cert\_file\_path) { ca\_cert\_file\_path\_ = ca\_cert\_file\_path; }

if (ca\_cert\_dir\_path) { ca\_cert\_dir\_path\_ = ca\_cert\_dir\_path; }

}

inline void SSLClient::enable\_server\_certificate\_verification(bool enabled) {

server\_certificate\_verification\_ = enabled;

}

inline long SSLClient::get\_openssl\_verify\_result() const {

return verify\_result\_;

}

inline SSL\_CTX \*SSLClient::ssl\_context() const noexcept { return ctx\_; }

inline bool SSLClient::process\_and\_close\_socket(

socket\_t sock, size\_t request\_count,

std::function<bool(Stream &strm, bool last\_connection,

bool &connection\_close)>

callback) {

request\_count = std::min(request\_count, keep\_alive\_max\_count\_);

return is\_valid() &&

detail::process\_and\_close\_socket\_ssl(

true, sock, request\_count, read\_timeout\_sec\_, read\_timeout\_usec\_,

ctx\_, ctx\_mutex\_,

[&](SSL \*ssl) {

if (ca\_cert\_file\_path\_.empty()) {

SSL\_CTX\_set\_verify(ctx\_, SSL\_VERIFY\_NONE, nullptr);

} else {

if (!SSL\_CTX\_load\_verify\_locations(

ctx\_, ca\_cert\_file\_path\_.c\_str(), nullptr)) {

return false;

}

SSL\_CTX\_set\_verify(ctx\_, SSL\_VERIFY\_PEER, nullptr);

}

if (SSL\_connect(ssl) != 1) { return false; }

if (server\_certificate\_verification\_) {

verify\_result\_ = SSL\_get\_verify\_result(ssl);

if (verify\_result\_ != X509\_V\_OK) { return false; }

auto server\_cert = SSL\_get\_peer\_certificate(ssl);

if (server\_cert == nullptr) { return false; }

if (!verify\_host(server\_cert)) {

X509\_free(server\_cert);

return false;

}

X509\_free(server\_cert);

}

return true;

},

[&](SSL \*ssl) {

SSL\_set\_tlsext\_host\_name(ssl, host\_.c\_str());

return true;

},

[&](SSL \* /\*ssl\*/, Stream &strm, bool last\_connection,

bool &connection\_close) {

return callback(strm, last\_connection, connection\_close);

});

}

inline bool SSLClient::is\_ssl() const { return true; }

inline bool SSLClient::verify\_host(X509 \*server\_cert) const {

/\* Quote from RFC2818 section 3.1 "Server Identity"

If a subjectAltName extension of type dNSName is present, that MUST

be used as the identity. Otherwise, the (most specific) Common Name

field in the Subject field of the certificate MUST be used. Although

the use of the Common Name is existing practice, it is deprecated and

Certification Authorities are encouraged to use the dNSName instead.

Matching is performed using the matching rules specified by

[RFC2459]. If more than one identity of a given type is present in

the certificate (e.g., more than one dNSName name, a match in any one

of the set is considered acceptable.) Names may contain the wildcard

character \* which is considered to match any single domain name

component or component fragment. E.g., \*.a.com matches foo.a.com but

not bar.foo.a.com. f\*.com matches foo.com but not bar.com.

In some cases, the URI is specified as an IP address rather than a

hostname. In this case, the iPAddress subjectAltName must be present

in the certificate and must exactly match the IP in the URI.

\*/

return verify\_host\_with\_subject\_alt\_name(server\_cert) ||

verify\_host\_with\_common\_name(server\_cert);

}

inline bool

SSLClient::verify\_host\_with\_subject\_alt\_name(X509 \*server\_cert) const {

auto ret = false;

auto type = GEN\_DNS;

struct in6\_addr addr6;

struct in\_addr addr;

size\_t addr\_len = 0;

#ifndef \_\_MINGW32\_\_

if (inet\_pton(AF\_INET6, host\_.c\_str(), &addr6)) {

type = GEN\_IPADD;

addr\_len = sizeof(struct in6\_addr);

} else if (inet\_pton(AF\_INET, host\_.c\_str(), &addr)) {

type = GEN\_IPADD;

addr\_len = sizeof(struct in\_addr);

}

#endif

auto alt\_names = static\_cast<const struct stack\_st\_GENERAL\_NAME \*>(

X509\_get\_ext\_d2i(server\_cert, NID\_subject\_alt\_name, nullptr, nullptr));

if (alt\_names) {

auto dsn\_matched = false;

auto ip\_mached = false;

auto count = sk\_GENERAL\_NAME\_num(alt\_names);

for (auto i = 0; i < count && !dsn\_matched; i++) {

auto val = sk\_GENERAL\_NAME\_value(alt\_names, i);

if (val->type == type) {

auto name = (const char \*)ASN1\_STRING\_get0\_data(val->d.ia5);

auto name\_len = (size\_t)ASN1\_STRING\_length(val->d.ia5);

if (strlen(name) == name\_len) {

switch (type) {

case GEN\_DNS: dsn\_matched = check\_host\_name(name, name\_len); break;

case GEN\_IPADD:

if (!memcmp(&addr6, name, addr\_len) ||

!memcmp(&addr, name, addr\_len)) {

ip\_mached = true;

}

break;

}

}

}

}

if (dsn\_matched || ip\_mached) { ret = true; }

}

GENERAL\_NAMES\_free((STACK\_OF(GENERAL\_NAME) \*)alt\_names);

return ret;

}

inline bool SSLClient::verify\_host\_with\_common\_name(X509 \*server\_cert) const {

const auto subject\_name = X509\_get\_subject\_name(server\_cert);

if (subject\_name != nullptr) {

char name[BUFSIZ];

auto name\_len = X509\_NAME\_get\_text\_by\_NID(subject\_name, NID\_commonName,

name, sizeof(name));

if (name\_len != -1) { return check\_host\_name(name, name\_len); }

}

return false;

}

inline bool SSLClient::check\_host\_name(const char \*pattern,

size\_t pattern\_len) const {

if (host\_.size() == pattern\_len && host\_ == pattern) { return true; }

// Wildcard match

// https://bugs.launchpad.net/ubuntu/+source/firefox-3.0/+bug/376484

std::vector<std::string> pattern\_components;

detail::split(&pattern[0], &pattern[pattern\_len], '.',

[&](const char \*b, const char \*e) {

pattern\_components.emplace\_back(std::string(b, e));

});

if (host\_components\_.size() != pattern\_components.size()) { return false; }

auto itr = pattern\_components.begin();

for (const auto &h : host\_components\_) {

auto &p = \*itr;

if (p != h && p != "\*") {

auto partial\_match = (p.size() > 0 && p[p.size() - 1] == '\*' &&

!p.compare(0, p.size() - 1, h));

if (!partial\_match) { return false; }

}

++itr;

}

return true;

}

#endif

} // namespace httplib

#endif // CPPHTTPLIB\_HTTPLIB\_H