Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE\_1\_0.txt or copy at <http://www.boost.org/LICENSE_1_0.txt>)

This library can be used to create applications that need to expose services through HTTP (e.g. embeddable ReST services).

该库可用于创建需要通过HTTP公开服务的应用程序（例如，可嵌入的ReST服务）。

|  |  |
| --- | --- |
| **WARNING** | This is not an official Boost C++ library. It wasn’t reviewed and can’t be downloaded from www.boost.org. [This library will be reviewed eventually](http://www.boost.org/community/review_schedule.html).  这不是官方的Boost C++库。未经审核，无法从www.boost.org下载。  这个库最终会被审核。 |

概念上Boost.Http是一个HTTP的Boost.Asio socket，这意味着可以利用到了像 SSL 和 coroutines等Boost.Asio特性，且与Boost.Asio sockets无缝集成。

Among its features:

* 支持现代HTTP特性
  + 管道和持久连接，其中相同的连接可用于服务HTTP多个请求
  + 分块（Chunking），其中HTTP主体可以被分割成更小的部分，然后逐个发送。这对于构建事件侦听器非常有用。
  + 升级，其中HTTP协议可以更改为另一个协议，如WebSockets
  + 安全性，其中SSL用于加密通信量
  + 100状态码 继续继续状态
* 细分api，统一HTTP消息和HTTP数据块
  + 您可以在元数据可用时立即处理请求（请求行加上标头部分）
  + 只有一个API可以获取请求体。此API允许渐进式下载。
  + 可以为普通的HTTP响应完整的body，也可以使用HTTP分块将其拆分为更小的部分
* 继承自ASIO的可扩展异步设计
  + 通过期货、协程或回调进行交互。Interaction via futures, coroutines, or callbacks.
  + 各种HTTP服务器架构都是可能的，无论是多线程服务器，每个请求一个协同路由，还是完全单线程的
  + 与其他非HTTP ASIO服务无缝集成.

Boost.Http还提供了最好的HTTP解析抽象:

* + 简单
  + 便携式（C++ 03，很少有其他依赖项）
  + 与Ryan Dahl的HTTP解析器一样，该解析器不进行任何系统调用或分配。它也不缓冲数据
  + 您可以改变传递的缓冲区[1]
  + 它不会窃取应用程序的控制流[2]。
  + 将令牌匹配和解码作为单独的步骤[3]

Boost.Http also provides a couple of special-purpose server classes:

* Lightweight standalone server.
  + 底层套接字类型是可自定义的，因此您也可以使用queue\_socket [[4](https://boostgsoc14.github.io/boost.http/#_footnotedef_4)][[5](https://boostgsoc14.github.io/boost.http/#_footnotedef_5)].
* 灵活的静态文件服务器，支持条件请求、部分下载和ETag识别
  + 您可以使用默认的文件解析程序，也可以自己提供一个。
  + 您可以调整默认解析器
  + 您可以添加最后一个钩子，以便在提供文件之前自定义响应消息
  + 无状态抽象，依赖操作系统功能提供缓存

Boost.Http是为多个后端设计的，如HTTP/2和FastGCI，这些都可以在将来添加。您可以在编译时和运行时多态之间进行选择，有适配器提供。

1. Using

1.1. Requirements

这个库是用C++ 11编写的，需要一个C++ 11支持编译器。This library is written in C++11 and requires a C++11 supporting compiler.

如果可能，这个库会尝试重用标准库组件，但情况并非总是如此。例如，在ASIO中，std:：error\_代码不能透明地替换为使用代码的ASIO中的boost:：system:：error\_代码。This library tries to reuse standard library components when possible, but this is not always the case. std::error\_code, for instance, cannot be transparently replaced by boost::system::error\_code within ASIO-using code.

The full list of dependencies is:

* CMake 3.1.0 or newer [[6](https://boostgsoc14.github.io/boost.http/#_footnotedef_6)]. You can skip this requirement if you don’t intend to run the tests.
* Boost 1.66 or newer.
  + algorithm, container, string\_view, ASIO, date\_time and system [[7](https://boostgsoc14.github.io/boost.http/#_footnotedef_7)].
  + unit\_test\_framework [[8](https://boostgsoc14.github.io/boost.http/#_footnotedef_8)].
  + context and coroutine [[9](https://boostgsoc14.github.io/boost.http/#_footnotedef_9)].
  + filesystem [[10](https://boostgsoc14.github.io/boost.http/#_footnotedef_10)].
* [asciidoctor](http://asciidoctor.org/), [asciidoctor-diagram](https://github.com/asciidoctor/asciidoctor-diagram/), [PlantUML](http://plantuml.com/) and [syntrax](https://kevinpt.github.io/syntrax/) [[11](https://boostgsoc14.github.io/boost.http/#_footnotedef_11)]. You’ll also need [pandoc](http://pandoc.org/) if you want to generate the ePUB output.

2. Tutorial

2.1. Message framework

在本教程中，您将对HTTP进行一个简短而快速的介绍（如果您已经熟悉HTTP，可以跳过相关的部分，这样您就更容易深入理解了Boost.Http更快）。您还将逐步学习如何使用Boost.Http。In this tutorial, you’ll have a small and fast introduction to HTTP (if you’re already familiar with HTTP, you can skip the associated section and it’s more likely you’ll deeply understand Boost.Http faster). You’ll also learn incrementally how to build a proper handling of HTTP requests using Boost.Http.

第二步是如何组织您的应用程序并制作一个有用的应用程序，但是这第二步留给用户自己处理。A second step would be how to organize your application and make an useful application, but this second step is left for the user to tackle alone.

|  |  |  |
| --- | --- | --- |
| **WARNING** | | This tutorial assumes you’re already familiar with ASIO! |
| **NOTE** | 本教程将使用协同程序，以提高可读性，这是其他任何方法都做不到的。您可能会对使用替代异步模型来实现更低的内存使用率或其他东西感兴趣。当C++为协同程序添加适当的支持时，这不仅是最可读的方式，而且也是最有性能的。This tutorial will make use of coroutines, to increase readability like nothing else could do. You might be interested in the use of alternative asynchronous models to achieve lower memory usage or something else. When C++ adds proper support for coroutines, this will not only be the most readable way, but also the most performant. | |

2.1.1. HTTP

HTTP是一种在可扩展性方面非常出色的协议。它的1.1版本自1997年以来一直没有改变过，至今仍能支持非常有创意的应用程序。HTTP/2.0标准已经发布，但大多数不同之处都与高效的连接管理有关，唯一可以影响使用HTTP的应用程序的更高层的功能是HTTP推送，它用于推测性地将数据发送到服务器预期客户端需要的客户端。HTTP is a protocol that shines in extensibility. Its 1.1 version has been used unchanged since 1997 and has been able to power very creative applications to this date. An HTTP/2.0 standard has been released, but most of the differencies are related to efficient connection management and the only feature that can affect higher-level layers of an application making use of HTTP is the HTTP push, which is used to speculatively send data to a client that the server anticipates the client will need.

HTTP also provides means to upgrade a protocol for a running connection and the WebSocket protocol is gaining importance where HTTP is used the most, in the web.

HTTP是一种面向请求和应答消息交换的协议。每个请求都是独立的，因此协议具有无状态性。每个请求都有一个关联的动词和路径，用来**告诉谁该怎么做**。每个HTTP消息都有一个主体（二进制数据的有效负载）和相关的元数据（即multimap<string，string>）。HTTP响应还有一个关联的状态代码，用于指示请求的操作是否成功HTTP is a protocol that is oriented to exchange of request and reply messages. Each request is independent, hence the stateless nature of the protocol. Each request has an associated verb and path, used to tell **what** do to **who**. Every HTTP message has a body (a payload of binary data) and associated metadata (i.e. a multimap<string, string>). The HTTP response also has a status code associated which is used to indicate success of the requested action.

每个HTTP消息中携带的元数据被命名为HTTP报头。HTTP报头包含如何处理数据负载的信息，包括mime类型、语言等信息。HTTP请求消息可以包含一个令牌，该令牌用于将多个不同的请求与同一个客户端相关联，因此用户不需要为每个页面请求键入用户名和密码。The metadata carried in every HTTP message is named as HTTP headers. The HTTP headers carry information on how to handle the data payload, with info such as the mime type, language and more. HTTP request messages can contain a token which is used to associate multiple different requests with the same client, so the user doesn’t need to type username and password for every page request.

GET是最常见的HTTP动词，它具有检索资源的简单语义。GET is the most common HTTP verb and it has the simple semantic to retrieve the resource.

如果您只想从本地文件系统公开一堆文件，您可能会对async\_response\_transmit\_dir感兴趣，它将承担您的多项责任（例如，部分下载和一些避免损坏文件的基本机制）If you just want to expose a bunch of files from the local filesystem, you may be interested in [async\_response\_transmit\_dir](https://boostgsoc14.github.io/boost.http/" \l "async_response_transmit_dir), which will take several responsibilities from you (e.g. partial download and a few basic mechanisms to avoid corrupt files).

HTTP应用程序的通常设置是解析请求URL并根据path组件选择适当的处理程序。可选的中间件处理程序可以根据身份验证、资源使用情况等执行一些ACL。几乎总是使用辅助数据库。The usual setup for an HTTP application is to parse the request URL and choose an appropriate handler based on the path component. Optional middleware handlers can do some ACL based on authentication, resource usage and so on. An auxiliar database is almost always used.

2.1.2. Accepting new connections

*Remember, the journey of a thousand miles begins with the first step.*

首先，如果我们想使用栈式协程来处理非预定数量的并发连接，我们将编写一个需要编写的样板代码。First, we’re going to write the boilerplate code Asio require us to write if we want use stackful coroutines to handle a non predetermined number of concurrent connections.

#include <iostream>

#include <boost/asio/spawn.hpp>

#include <boost/http/buffered\_socket.hpp>

using namespace std;

using namespace boost;

class connection: public std::enable\_shared\_from\_this<connection>{

public:

void operator()(asio::yield\_context yield){

auto self = shared\_from\_this();

try {

cout << "--\n[" << self->counter << "] Socket ready" << endl;

// >>> OUR CODE GOES HERE <<<

} catch (system::system\_error &e) {

if (e.code() != system::error\_code{asio::error::eof}) {

cerr << '[' << self->counter << "] Aborting on exception: "

<< e.what() << endl;

std::exit(1);

}

cout << '[' << self->counter << "] Error: "

<< e.what() << endl;

} catch (std::exception &e) {

cerr << '[' << self->counter << "] Aborting on exception: "

<< e.what() << endl;

std::exit(1);

}

}

asio::ip::tcp::socket &tcp\_layer(){

return socket.next\_layer();

}

static std::shared\_ptr<connection> make\_connection(asio::io\_context &ios,

int counter) {

return std::shared\_ptr<connection>{new connection{ios, counter}};

}

private:

connection(asio::io\_context &ios, int counter)

: socket(ios)

, counter(counter)

{}

http::buffered\_socket socket;

int counter;

};

int main(){

asio::io\_context ios;

asio::ip::tcp::acceptor acceptor(ios, asio::ip::tcp

::endpoint(asio::ip::tcp::v6(), 8080));

auto do\_accept = [&acceptor,&ios](asio::yield\_context yield) {

int counter = 0;

for ( ; true ; ++counter ) {

try {

auto connection = connection::make\_connection(ios, counter);

cout << "About to accept a new connection" << endl;

acceptor.async\_accept(connection->tcp\_layer(), yield);

auto handle\_connection = [connection](asio::yield\_context yield) mutable {

(\*connection)(yield);

};

spawn(acceptor.get\_executor(), handle\_connection);

} catch (std::exception &e) {

cerr << "Aborting on exception: " << e.what() << endl;

std::exit(1);

}

}

};

cout << "About to schedule new connections acceptance" << endl;

spawn(ios, do\_accept);

cout << "About to run the I/O scheduler/executor" << endl;

ios.run();

return 0;

}

总之，我们使用共享的ptr来分配对象，并确保只要有对它的引用，它就会保持活动状态，并且我们生成了一个acceptor算法，该算法将为每个连接创建一个 shared\_ptr。Summarizing, we make use of shared\_ptr to alloc object and ensure it’ll stay alive as long as there is some reference to it and we spawn an acceptor algorithm who will create a shared\_ptr for every connection.

asio::ip::tcp::socket &connection::tcp\_layer(){

return socket.next\_layer();

}

The basic\_buffered\_socket<T>::next\_layer() member-function will return the internal T object. The buffered\_socket typedef uses asio::ip::tcp::socket as T.

spawn(acceptor.get\_executor(), handle\_connection);

We spawn a new handler for every connection, so we’ll be able to handle them all concurrently.

auto self = shared\_from\_this();

We must create a reference to the shared\_ptr before calling any asynchronous function to ensure the object will be live as long as the coroutine.

2.1.3. Receiving requests

你可以在本段末尾找到全部代码。现在，我们在前面代码的基础上构建代码（只需将“我们的代码转到”这里的注释替换为我们将在本节中构建的代码。You can find the whole code at the end of the secion. For now, we build upon the code from the previous code (just replace the “our code goes” here comment with the code we’ll build in this section.

while (self->socket.is\_open()) {

因此，您首先应该在套接字打开时执行循环，这样就可以处理请求的整个管道。如果连接非正常关闭，则在其中一个操作期间将生成一个错误代码（由协同路由完成令牌转换为异常）。如果连接正常关闭，循环最终将通过返回false的is\_open（）停止。So, the first thing you should do is loop while the socket is open, so the whole pipelining of requests will be handled. If the connection closes ungracefully, an error code will be generated (converted to exceptions by the coroutine completion token) during one of the operations. If the connection closes gracefully, the loop will eventually stop by is\_open() returning false.

self->socket.async\_read\_request(self->request, yield);

因此，实际处理的第一部分是请求请求消息。在调用完成处理程序时，您将获得完整的方法和路径（对于协同程序，这意味着下一行代码）。当操作正在进行时，您不能接触这些变量中的任何一个（如果使用协同程序，则很难出错）。self→request→headers（）也将被填充，以及主体中已经接收到的任何部分。So, the first part to actually handle is to ask for the request message. You’ll get the full method and path by the time the completion handler is called (in the case of coroutines, this means the next line of code). You must not touch any of these variables while the operation is in progress (hard to get it wrong if you’re using coroutines). self→request→headers() will also be filled and whatever part of the body that has already been received.

while (self->socket.read\_state() != http::read\_state::empty) {

// ...

switch (self->socket.read\_state()) {

case http::read\_state::message\_ready:

// ...

self->socket.async\_read\_some(self->request, yield);

break;

case http::read\_state::body\_ready:

// ...

self->socket.async\_read\_trailers(self->request, yield);

break;

You can use self→socket.read\_state() to know which part of the request is still missing and ask for the rest.

But there is a little gotcha. You should send a 100-continue to ask for the rest of the body if required. This feature appeared first in HTTP/1.1 as a mean to better use network traffic, by allowing you to reject requests sooner.

if (http::request\_continue\_required(self->request)) {

// ...

self->socket.async\_write\_response\_continue(yield);

}

This is how we check if we should send 100-continue. It must be done after async\_read\_request and before async\_read\_some.

By this time, we have already fully received the request and we can do something with it. Pretty easy, see?

http::response reply;

The response message we’re about to send.

std::string body{"Hello World from \""};

body += self->request.target();

body += "\"\n";

std::copy(body.begin(), body.end(),

std::back\_inserter(reply.body()));

We feed some bytes to the body.

reply.status\_code() = 200;

reply.reason\_phrase() = "OK";

self->socket.async\_write\_response(reply, yield);

And then we send the response. Our application is complete.

There is a gotcha here. If you use pure asio::ip::tcp::socket, you’re subject to Asio composed operations restrictions and you should not schedule any operation while the previous one is in progress. You can wrap asio::ip::tcp::socket into some queuing socket [[12](https://boostgsoc14.github.io/boost.http/#_footnotedef_12)] to work around this bug and Boost.Http will give you the required customization points to allow you to use it. We don’t worry about this problem here, because with coroutines we’re done.

if (we\_are\_halting())

reply.headers().emplace("connection", "close");

If we’re going to halt the server and want to gracefully close current connections, this code can be used to close the HTTP pipeline after the current response end. You should pay attention to safe and idempotent methods if you want to learn more about HTTP pipelining and HTTP in general.

self->request.body().clear(); // free unused resources

Given we’re consuming the body, it’s a good idea to free unused resources. We don’t use C++11’s shrink\_to\_fit because it could trigger another reallocation in the next piece of body received. The idea is to reuse a small allocated buffer instead. You could also create your own message type to discard feeded bytes.

acceptor.async\_accept(connection->tcp\_layer(), yield);

HTTP doesn’t require you to handle protocol negotiation separately from the remaining protocol or any super special handshaking flow. Therefore, we use a “naked” acceptor to fuel an usable HTTP socket. Other HTTP backends may have different usage.

And, like I promised, here is the complete code (with a lot of print statements and a few lines to demonstrate more usage):

#include <iostream>

#include <boost/asio/spawn.hpp>

#include <boost/http/buffered\_socket.hpp>

#include <boost/http/algorithm/query.hpp>

#include <boost/http/request.hpp>

#include <boost/http/response.hpp>

using namespace std;

using namespace boost;

bool we\_are\_halting()

{

return false;

}

class connection: public std::enable\_shared\_from\_this<connection>

{

public:

void operator()(asio::yield\_context yield)

{

auto self = shared\_from\_this();

try {

cout << "[" << self->counter << "] Socket ready" << endl;

while (self->socket.is\_open()) {

cout << "--\n[" << self->counter

<< "] About to receive a new message" << endl;

self->socket.async\_read\_request(self->request, yield);

self->request.body().clear(); // free unused resources

if (http::request\_continue\_required(self->request)) {

cout << '[' << self->counter

<< "] Continue required. About to send"

" \"100-continue\""

<< std::endl;

self->socket.async\_write\_response\_continue(yield);

}

while (self->socket.read\_state() != http::read\_state::empty) {

cout << '[' << self->counter

<< "] Message not fully received" << endl;

switch (self->socket.read\_state()) {

case http::read\_state::message\_ready:

cout << '[' << self->counter

<< "] About to receive some body" << endl;

self->socket.async\_read\_some(self->request, yield);

self->request.body().clear(); // free unused resources

break;

case http::read\_state::body\_ready:

cout << '[' << self->counter

<< "] About to receive trailers" << endl;

self->socket.async\_read\_trailers(self->request, yield);

self->request.body().clear(); // free unused resources

break;

default:;

}

}

cout << '[' << self->counter << "] Message received. State = "

<< int(self->socket.read\_state()) << endl;

cout << '[' << self->counter << "] Method: "

<< self->request.method() << endl;

cout << '[' << self->counter << "] Path: "

<< self->request.target() << endl;

{

auto host = self->request.headers().find("host");

if (host != self->request.headers().end())

cout << '[' << self->counter << "] Host header: "

<< host->second << endl;

}

std::cout << '[' << self->counter << "] Write state = "

<< int(self->socket.write\_state()) << std::endl;

cout << '[' << self->counter << "] About to send a reply"

<< endl;

http::response reply;

if (we\_are\_halting())

reply.headers().emplace("connection", "close");

std::string body{"Hello World from \""};

body += self->request.target();

body += "\"\n";

std::copy(body.begin(), body.end(),

std::back\_inserter(reply.body()));

reply.status\_code() = 200;

reply.reason\_phrase() = "OK";

self->socket.async\_write\_response(reply, yield);

}

} catch (system::system\_error &e) {

if (e.code() != system::error\_code{asio::error::eof}) {

cerr << '[' << self->counter << "] Aborting on exception: "

<< e.what() << endl;

std::exit(1);

}

cout << '[' << self->counter << "] Error: "

<< e.what() << endl;

} catch (std::exception &e) {

cerr << '[' << self->counter << "] Aborting on exception: "

<< e.what() << endl;

std::exit(1);

}

}

asio::ip::tcp::socket &tcp\_layer()

{

return socket.next\_layer();

}

static std::shared\_ptr<connection> make\_connection(asio::io\_context &ios,

int counter)

{

return std::shared\_ptr<connection>{new connection{ios, counter}};

}

private:

connection(asio::io\_context &ios, int counter)

: socket(ios)

, counter(counter)

{}

http::buffered\_socket socket;

int counter;

http::request request;

};

int main()

{

asio::io\_context ios;

asio::ip::tcp::acceptor acceptor(ios,

asio::ip::tcp

::endpoint(asio::ip::tcp::v6(), 8080));

auto do\_accept = [&acceptor,&ios](asio::yield\_context yield) {

int counter = 0;

for ( ; true ; ++counter ) {

try {

auto connection

= connection::make\_connection(ios, counter);

cout << "About to accept a new connection" << endl;

acceptor.async\_accept(connection->tcp\_layer(), yield);

auto handle\_connection

= [connection](asio::yield\_context yield) mutable {

(\*connection)(yield);

};

spawn(acceptor.get\_executor(), handle\_connection);

} catch (std::exception &e) {

cerr << "Aborting on exception: " << e.what() << endl;

std::exit(1);

}

}

};

cout << "About to schedule new connections acceptance" << endl;

spawn(ios, do\_accept);

cout << "About to run the I/O scheduler/executor" << endl;

ios.run();

return 0;

}

2.1.4. Using chunked messages

In the previous example, we used atomic messages to respond the request. But this is limiting when we’re trying to achieve certain kind of tasks, like serving a live video stream. A second option for us is to use chunked messages to respond the request.

|  |  |
| --- | --- |
| **WARNING** | Chunked messages are not always available and you must check if you can use chunked messages for every request with the write\_response\_native\_stream() socket member-function. |

If chunked messages are available, you can use the following sequence of actions to respond the request:

1. async\_write\_response\_metadata once.
2. async\_write zero or more times.
3. async\_write\_trailers or async\_write\_end\_of\_message, once.

2.1.5. Runtime-based polymorphic abstractions

If you want to create a function that will handle requests originated from different HTTP backends, you have three choices:

* Rewrite the handler for every HTTP backend.
* Write generic handlers.
* Type-erase the HTTP backends.

Boost.Http provide some abstractions to type erase the HTTP backends (playing a similar role to std::function). The starting point to learn about it is the [server\_socket\_adaptor page](https://boostgsoc14.github.io/boost.http/" \l "server_socket_adaptor).

void handler(http::poly\_server\_socket &socket)

{

// ...

}

http::server\_socket\_adaptor<http::buffered\_socket>

socket(acceptor.get\_executor().context());

handler(socket);

2.2. Polymorphic handlers (std::function alternative)

Boost.Asio uses handlers with signatures similar to the following to notify about the completion of tasks:

void(boost::system::error\_code)

If you intend to wrap the template heavy usage of Boost.Asio behind a stable interface that can be accessed through dynamically loaded plug-ins, you may be tempted to use std::function:

class my\_socket

{

public:

virtual ~my\_socket() = default;

virtual void async\_write\_some(

boost::asio::const\_buffer buf,

std::function<void(boost::system::error\_code, std::size\_t)> handler

) = 0;

virtual void async\_read\_some(

std::vector<char> &buf,

std::function<void(boost::system::error\_code, std::size\_t)> handler

) = 0;

};

void work()

{

// ...

auto handle\_socket = boost::dll::import<void(std::shared\_ptr<my\_socket>)>(

path\_to\_shared\_library, "handle\_socket"

);

// ...

for (;;) {

// ...

handle\_socket(sock);

}

}

However, this approach is wrong. std::function does type erasure of functors, but it won’t do type erasure of associated allocators and associated executors which are a core part of Boost.Asio. This extra information will be discarded.

If there is an associated executor with a completion handler, this associated executor should be used and propagated by every intermediate handler in the chain.

If I try the following, the propagation will stop at the my\_socket boundary as std::function discards the associated executor:

void work()

{

// ...

auto handler = boost::asio::bind\_executor(

boost::asio::io\_context::strand{ctx},

[](boost::system::error\_code, std::size\_t) {}

);

boost::asio::async\_write(\*sock, handler);

}

One way to verify this assertion is with a simple test case. Create two execution contexts, ctx1 and ctx2. Post a handler to ctx1 and then call ctx1.run(). You should expect the handler to be called. Now modify handler to be wrapped and associated with an executor from ctx2. If only this change is done, you should expect the program to exit without ever calling the handler. The following code illustrates this situation:

#include <iostream>

#include <boost/asio/bind\_executor.hpp>

#include <boost/asio/io\_context.hpp>

#include <boost/asio/post.hpp>

int main()

{

boost::asio::io\_context ctx1;

boost::asio::io\_context ctx2;

auto handler = boost::asio::bind\_executor(

ctx2,

[]() {

// won't happen

std::cout << "handler called" << std::endl;

}

);

boost::asio::post(ctx1, handler);

ctx1.run();

return 0;

}

If you also change the code to call ctx2.run() after ctx1.run(), you should expect the handler to be called. But this is just an exercise for the reader. We’re sticking with the previous code of no handler called as the desired behaviour. To be more specific, we’re interested in making sure that we can *observe* that execution scheduling is done through a specific executor (i.e. associated executors are respected). And, for this executor in particular, the observed output is of no handler called.

If you wrap our handler within a std::function object, the associated executors will be discarded and you should expect the handler to be called again:

#include <functional>

#include <iostream>

#include <boost/asio/bind\_executor.hpp>

#include <boost/asio/io\_context.hpp>

#include <boost/asio/post.hpp>

int main()

{

boost::asio::io\_context ctx1;

boost::asio::io\_context ctx2;

auto handler = boost::asio::bind\_executor(

ctx2,

[]() {

std::cout << "handler called" << std::endl;

}

);

boost::asio::post(ctx1, std::function<void()>{handler});

ctx1.run();

return 0;

}

That’s why we provide boost::http::asio::experimental::poly\_handler as an alternative to std::function:

#include <iostream>

#include <boost/http/asio/experimental/poly\_handler.hpp>

#include <boost/asio/bind\_executor.hpp>

#include <boost/asio/io\_context.hpp>

#include <boost/asio/post.hpp>

int main()

{

boost::asio::io\_context ctx1;

boost::asio::io\_context ctx2;

auto handler = boost::asio::bind\_executor(

ctx2,

[]() {

// won't happen

std::cout << "handler called" << std::endl;

}

);

boost::asio::post(

ctx1,

boost::http::asio::experimental::poly\_handler<void()>{handler}

);

ctx1.run();

return 0;

}

If you call boost::asio::get\_associated\_executor on boost::http::asio::experimental::poly\_handler, you’ll get a type erased (i.e. boost::asio::executor) executor:

auto phandler

= boost::http::asio::experimental::poly\_handler<void()>{handler};

auto ex = boost::asio::get\_associated\_executor(phandler);

Using poly\_handler, we can update the previous my\_socket definition and make the DLL boundary work properly:

class my\_socket

{

public:

virtual ~my\_socket() = default;

virtual void async\_write\_some(

boost::asio::const\_buffer buf,

boost::http::asio::experimental::poly\_handler<

void(boost::system::error\_code, std::size\_t)

> handler

) = 0;

virtual void async\_read\_some(

std::vector<char> &buf,

boost::http::asio::experimental::poly\_handler<

void(boost::system::error\_code, std::size\_t)

> handler

) = 0;

};

poly\_handler will also preserve the associated allocators as can be verified with the following example:

#include <iostream>

#include <boost/http/asio/experimental/poly\_handler.hpp>

struct MyHandler

{

using allocator\_type = std::experimental::pmr::polymorphic\_allocator<void>;

MyHandler(std::experimental::pmr::memory\_resource \*r)

: r(r)

{}

void operator()() {}

allocator\_type get\_allocator() const

{

return {r};

}

std::experimental::pmr::memory\_resource \*r;

};

int main()

{

auto r1 = std::experimental::pmr::new\_delete\_resource();

auto r2 = std::experimental::pmr::null\_memory\_resource();

boost::http::asio::experimental::poly\_handler<void()>

handler{MyHandler{r1}};

std::experimental::pmr::polymorphic\_allocator<void> fallback\_a = r2;

auto a = boost::asio::get\_associated\_allocator(handler, fallback\_a);

std::cout << "r1 = " << (void\*)(r1) << std::endl;

std::cout << "r2 = " << (void\*)(r2) << std::endl;

std::cout << "a = " << (void\*)(a.resource()) << std::endl;

return 0;

}

In the example, we associated the MyHandler object with r1 just to wrap it under poly\_handler. Then we asked what was the associated allocator a and passed r2 as the fallback in case there was no such a. The expected output would be for r1 to be the same as a. And that’s just what you’ll see if you run the example. If you want the example to fail (as in discarding the associated allocator and showing a different output), just replace poly\_handler with std::function.

|  |  |
| --- | --- |
| **IMPORTANT** | If you try to use an associated allocator whose type doesn’t match std::experimental::pmr::polymorphic\_allocator<T>, poly\_handler will fail to wrap it and it won’t compile. This behaviour is on purpose as we should not silently discard associated properties. The reason why I don’t support other types of allocators for associated allocators (fallback allocators passed to boost::asio::get\_associated\_allocator can still be of different types) is because they make little sense in a generic context such as ASIO’s and implementation complexity. |

2.3. Parsing (beginner)

In this tutorial, you’ll learn how to use this library to parse HTTP streams easily.

|  |  |
| --- | --- |
| **NOTE** | We assume the reader has basic understanding of C++ and Boost.Asio. |

We start with the code that should resemble the structure of the program you’re about to code. And this structure is as follows:

#include <boost/http/reader/request.hpp>

#include <string>

#include <map>

namespace http = boost::http;

namespace asio = boost::asio;

struct my\_socket\_consumer

{

private:

http::reader::request request\_reader;

std::string buffer;

std::string last\_header;

public:

std::string method;

std::string request\_target;

int version;

std::multimap<std::string, std::string> headers;

void on\_socket\_callback(asio::buffer data)

{

using namespace http::token;

using token::code;

buffer.push\_back(data);

request\_reader.set\_buffer(buffer);

while (request\_reader.code() != code::end\_of\_message) {

switch (request\_reader.code()) {

case code::skip:

// do nothing

break;

case code::method:

method = request\_reader.value<token::method>();

break;

case code::request\_target:

request\_target = request\_reader.value<token::request\_target>();

break;

case code::version:

version = request\_reader.value<token::version>();

break;

case code::field\_name:

case code::trailer\_name:

last\_header = request\_reader.value<token::field\_name>();

}

request\_reader.next();

}

request\_reader.next();

ready();

}

protected:

virtual void ready() = 0;

};

You’re building a piece of code that consumes HTTP from somewhere — the in — and spits it out in the form of C++ structured data to somewhere else — the out.

The *in* of your program is connected to the above piece of code through the on\_socket\_callback member function. The *out* of your program is connected to the previous piece of code through the ready overrideable member-function.

By now I shouldn’t be worried about your understanding of how you’ll connect the network I/O with the *in* of the program. The connection point is obvious by now. However, I’ll briefly explain the *out* connection point and then we can proceed to delve into the *inout*-inbetween (Danas) part of the program.

Once the ready member function is called, the data for your request will be available in the method, request\_target and the other *public* variables. From now on, I’ll focus my attention to the sole implementation of my\_socket\_consumer::on\_socket\_callback.

*The awaited prize*

void my\_socket\_consumer::on\_socket\_callback(asio::buffer data)

{

//http::reader::request request\_reader;

//std::string buffer;

//std::string last\_header;

using namespace http::token;

using token::code;

buffer.push\_back(data);

request\_reader.set\_buffer(buffer);

while (request\_reader.code() != code::end\_of\_message) {

switch (request\_reader.code()) {

case code::skip:

// do nothing

break;

case code::method:

method = request\_reader.value<token::method>();

break;

case code::request\_target:

request\_target = request\_reader.value<token::request\_target>();

break;

case code::version:

version = request\_reader.value<token::version>();

break;

case code::field\_name:

case code::trailer\_name:

last\_header = request\_reader.value<token::field\_name>();

}

request\_reader.next();

}

request\_reader.next();

ready();

}

Try to keep in mind the three variables that will really orchestrate the flow: request\_reader, buffer and last\_header.

The whole work is about managing the buffer and managing the tokens.

The token access is very easy. As the parser is incremental, there is only one token at a time. I don’t need to explain Boost.Http control-flow because the control flow will be coded by you (a library, not a framework). You only have to use code() to check the current token and value<T>() to extract its value. Use next() to advance a token.

|  |  |
| --- | --- |
| **WARNING** | There is only one caveat. The parser doesn’t buffer data and will decode the token into a value (the value<T>() member function) directly from the buffer data.  This means you cannot extract the current value once you drop current buffer data. As a nice side effect, you spare CPU time for the tokens you do not need to decode (match’n’decoding as separate steps). |

The parser doesn’t buffer data, which means when we use the set\_buffer member function, request\_reader only maintains a **view** to the passed buffer, which we’ll refer to as the virtual buffer from now on.

In the virtual buffer, there is head/current and remaining/tail. request\_reader doesn’t store a pointer/address/index to the real buffer. Once a token is consumed, his bytes (head) are discarded from the virtual buffer. When you mutate the real buffer, the virtual buffer is invalidated and you must inform the parser using set\_buffer. However, the bytes discarded from the virtual buffer shouldn’t appear again. You must keep track of the number of discarded bytes to prepare the buffer to the next call to set\_buffer. The previous code doesn’t handle that.

The new tool that you should be presented now is token\_size(). token\_size() will return the size in bytes of current/head.

|  |  |
| --- | --- |
| **WARNING** | There is no guarantee token\_size() returns the same size as returned by string\_length(request\_reader.value<T>()). You **need** to use token\_size() to compute the number of discarded bytes. |

void my\_socket\_consumer::on\_socket\_callback(asio::buffer data)

{

using namespace http::token;

using token::code;

buffer.push\_back(data);

request\_reader.set\_buffer(buffer);

std::size\_t nparsed = 0; //< NEW

while (request\_reader.code() != code::end\_of\_message) {

switch (request\_reader.code()) {

case code::skip:

// do nothing

break;

case code::method:

method = request\_reader.value<token::method>();

break;

case code::request\_target:

request\_target = request\_reader.value<token::request\_target>();

break;

case code::version:

version = request\_reader.value<token::version>();

break;

case code::field\_name:

case code::trailer\_name:

last\_header = request\_reader.value<token::field\_name>();

}

nparsed += request\_reader.token\_size(); //< NEW

request\_reader.next();

}

nparsed += request\_reader.token\_size(); //< NEW

request\_reader.next();

buffer.erase(0, nparsed); //< NEW

ready();

}

nparsed was easy. However, the while(request\_reader.code() != code::end\_of\_message) doesn’t seem right. It’s very error-prone to assume the full HTTP message will be ready in a single call to on\_socket\_callback. Error handling must be introduced in the code.

void my\_socket\_consumer::on\_socket\_callback(asio::buffer data)

{

using namespace http::token;

using token::code;

buffer.push\_back(data);

request\_reader.set\_buffer(buffer);

std::size\_t nparsed = 0;

while (request\_reader.code() != code::error\_insufficient\_data //< NEW

&& request\_reader.code() != code::end\_of\_message) { //< NEW

switch (request\_reader.code()) {

case code::skip:

// do nothing

break;

case code::method:

method = request\_reader.value<token::method>();

break;

case code::request\_target:

request\_target = request\_reader.value<token::request\_target>();

break;

case code::version:

version = request\_reader.value<token::version>();

break;

case code::field\_name:

case code::trailer\_name:

last\_header = request\_reader.value<token::field\_name>();

}

nparsed += request\_reader.token\_size();

request\_reader.next();

}

nparsed += request\_reader.token\_size();

request\_reader.next();

buffer.erase(0, nparsed);

if (request\_reader.code() == code::error\_insufficient\_data) //< NEW

return; //< NEW

ready();

}

|  |  |
| --- | --- |
| **NOTE** | Don’t worry about token\_size(code::error\_insufficient\_data) being added to nparsed. This (error) "token" is defined to be 0-size (it fits perfectly with the other rules). |

Just because it’s easy and we’re already at it, let’s handle the other errors as well:

void my\_socket\_consumer::on\_socket\_callback(asio::buffer data)

{

using namespace http::token;

using token::code;

buffer.push\_back(data);

request\_reader.set\_buffer(buffer);

std::size\_t nparsed = 0;

while (request\_reader.code() != code::error\_insufficient\_data

&& request\_reader.code() != code::end\_of\_message) {

switch (request\_reader.code()) {

case code::error\_set\_method: //< NEW

case code::error\_use\_another\_connection: //< NEW

// Can only happen in response parsing code.

assert(false); //< NEW

case code::error\_invalid\_data: //< NEW

case code::error\_no\_host: //< NEW

case code::error\_invalid\_content\_length: //< NEW

case code::error\_content\_length\_overflow: //< NEW

case code::error\_invalid\_transfer\_encoding: //< NEW

case code::error\_chunk\_size\_overflow: //< NEW

throw "invalid HTTP data"; //< NEW

case code::skip:

// do nothing

break;

case code::method:

method = request\_reader.value<token::method>();

break;

case code::request\_target:

request\_target = request\_reader.value<token::request\_target>();

break;

case code::version:

version = request\_reader.value<token::version>();

break;

case code::field\_name:

case code::trailer\_name:

last\_header = request\_reader.value<token::field\_name>();

}

nparsed += request\_reader.token\_size();

request\_reader.next();

}

nparsed += request\_reader.token\_size();

request\_reader.next();

buffer.erase(0, nparsed);

if (request\_reader.code() == code::error\_insufficient\_data)

return;

ready();

}

And buffer management is complete. However, the code only demonstrated how to extract simple tokens. Field name and field value are simple tokens, but they are usually tied together into a complex structure.

void my\_socket\_consumer::on\_socket\_callback(asio::buffer data)

{

using namespace http::token;

using token::code;

buffer.push\_back(data);

request\_reader.set\_buffer(buffer);

std::size\_t nparsed = 0;

while (request\_reader.code() != code::error\_insufficient\_data

&& request\_reader.code() != code::end\_of\_message) {

switch (request\_reader.code()) {

// ...

case code::skip:

break;

case code::method:

method = request\_reader.value<token::method>();

break;

case code::request\_target:

request\_target = request\_reader.value<token::request\_target>();

break;

case code::version:

version = request\_reader.value<token::version>();

break;

case code::field\_name:

case code::trailer\_name:

last\_header = request\_reader.value<token::field\_name>();

break;

case code::field\_value: //< NEW

case code::trailer\_value: //< NEW

// NEW

headers.emplace(last\_header,

request\_reader.value<token::field\_value>());

}

nparsed += request\_reader.token\_size();

request\_reader.next();

}

nparsed += request\_reader.token\_size();

request\_reader.next();

buffer.erase(0, nparsed);

if (request\_reader.code() == code::error\_insufficient\_data)

return;

ready();

}

last\_header did the trick. Easy, but maybe we want to separate headers and trailers (the HTTP headers that are sent *after* the message body). This task can be accomplished by the use of structural tokens.

void my\_socket\_consumer::on\_socket\_callback(asio::buffer data)

{

// NEW:

// We have to declare `bool my\_socket\_consumer::use\_trailers = false` and

// `std::multimap<std::string, std::string> my\_socket\_consumer::trailers`.

using namespace http::token;

using token::code;

buffer.push\_back(data);

request\_reader.set\_buffer(buffer);

std::size\_t nparsed = 0;

while (request\_reader.code() != code::error\_insufficient\_data

&& request\_reader.code() != code::end\_of\_message) {

switch (request\_reader.code()) {

// ...

case code::skip:

break;

case code::method:

method = request\_reader.value<token::method>();

break;

case code::request\_target:

request\_target = request\_reader.value<token::request\_target>();

break;

case code::version:

version = request\_reader.value<token::version>();

break;

case code::field\_name:

case code::trailer\_name:

last\_header = request\_reader.value<token::field\_name>();

break;

case code::field\_value:

case code::trailer\_value:

// NEW

(use\_trailers ? trailers : headers)

.emplace(last\_header,

request\_reader.value<token::field\_value>());

break;

case code::end\_of\_headers: //< NEW

use\_trailers = true; //< NEW

}

nparsed += request\_reader.token\_size();

request\_reader.next();

}

nparsed += request\_reader.token\_size();

request\_reader.next();

buffer.erase(0, nparsed);

if (request\_reader.code() == code::error\_insufficient\_data)

return;

ready();

}

|  |  |
| --- | --- |
| **NOTE** | Maybe you had a gut feeling and thought that the previous code was too strange. If trailer\_name is a separate token, why don’t we use request\_reader.value<token::trailer\_name>() (same to trailer\_value) and go away with structural tokens?  Yes, I unnecessarily complicated the code here to introduce you the concept of structural tokens. They are very important and usually you’ll end up using them. Maybe this tutorial needs some revamping after the library evolved a few times.  Also notice that here you can use either request\_reader.value<token::field\_name>() or request\_reader.value<token::trailer\_name>() to extract this token value. It is as if trailer\_name is “implicitly convertible” to field\_name, so to speak. This feature makes the life of users who don’t need to differentiate headers and trailers much easier (with no drawback to the users who do need to differentiate them). |

Some of the structural tokens' properties are:

* No value<T>() associated. value<T>() extraction is a property exclusive of the *data* tokens.
* It might be 0-sized.
* They are always emitted (e.g. code::end\_of\_body will be emitted before code::end\_of\_message even if no code::body\_chunk is present).

We were using the code::end\_of\_message structural token since the initial version of the code, so they aren’t completely alien. However, we were ignoring one very important HTTP parsing feature for this time. It’s the last missing bit before your understanding to use this library is complete. Our current code lacks the ability to handle HTTP pipelining.

HTTP pipelining is the feature that allows HTTP clients to send HTTP requests “in batch”. In other words, they may send several requests at once over the same connection before the server creates a response to them. If the previous code faces this situation, it’ll stop parsing on the first request and possibly wait forever until the on\_socket\_callback is called again with more data (yeap, networking code can be hard with so many little details).

void my\_socket\_consumer::on\_socket\_callback(asio::buffer data)

{

using namespace http::token;

using token::code;

buffer.push\_back(data);

request\_reader.set\_buffer(buffer);

std::size\_t nparsed = 0;

while (request\_reader.code() != code::error\_insufficient\_data

&& request\_reader.code() != code::end\_of\_message) {

switch (request\_reader.code()) {

// ...

case code::skip:

break;

case code::method:

use\_trailers = false; //< NEW

headers.clear(); //< NEW

trailers.clear(); //< NEW

method = request\_reader.value<token::method>();

break;

case code::request\_target:

request\_target = request\_reader.value<token::request\_target>();

break;

case code::version:

version = request\_reader.value<token::version>();

break;

case code::field\_name:

case code::trailer\_name:

last\_header = request\_reader.value<token::field\_name>();

break;

case code::field\_value:

case code::trailer\_value:

(use\_trailers ? trailers : headers)

.emplace(last\_header,

request\_reader.value<token::field\_value>());

break;

case code::end\_of\_headers:

use\_trailers = true;

}

nparsed += request\_reader.token\_size();

request\_reader.next();

}

nparsed += request\_reader.token\_size();

request\_reader.next();

buffer.erase(0, nparsed);

if (request\_reader.code() == code::error\_insufficient\_data)

return;

ready();

if (buffer.size() > 0) //< NEW

on\_socket\_callback(); //< NEW

}

There are HTTP libraries that could adopt a “synchronous” approach where the user must immediately give a HTTP response once the ready() callback is called so the parsing code can parse the whole buffer until the end and we could just put the ready() call into the code::end\_of\_message case.

There are HTTP libraries that follow ASIO active style and we expect the user to call something like async\_read\_request before it can read the next request. In this case, the solution for HTTP pipelining would be different.

There are libraries that don’t follow ASIO style, but don’t force the user to send HTTP responses immediately on the ready() callback. In such cases, synchronization/coordination of the response generation by the user and parse resuming by the library is necessary.

This point can be rather diverse and the code for this tutorial only shows a rather quick’n’dirty solution. Any different solution to keep the parsing train at full-speed is left as an exercise to the reader.

The interesting point about the code here is to clear the state of the *to-be-parsed* message before each request-response pair. In the previous code, this was done binding the “method token arrived” event — the first token in a HTTP request — with such state cleanup.

By now, you’re ready to use this library in your projects. You may want to check Boost.Http own usage of the parser or the Tufão library as real-world and complete examples of this parser.

*See also*

* [What are the differences between reader::request and reader::response?](https://boostgsoc14.github.io/boost.http/#request_response_diff)

2.4. Parsing (advanced)

In this tutorial, you’ll learn how to use this library to parse HTTP streams easily.

The architecture of the library is broken into two classes of parsers, the content parsers and the structural parsers.

The content parsers handle non-structural elements, terminal tokens, all easy to match and decode. They are stateless. They are mini-parsers for elements easy to parse and by themselves don’t add much value to justify a library (don’t confuse *low value* with *valueless*). They live in the boost::http::syntax namespace. They are useful when you want to parse individual HTTP elements like the range header value. We won’t see them in this tutorial.

The structural parsers handle structured data formats (e.g. HTTP). To achieve flexibility and performance requirements, they follow the incremental/pull parser model (a bit like the more traditional Iterator design pattern as described in the Gang-of-Four book, instead of C++ iterators). These parsers live in the boost::http::reader namespace. These are the parsers we will look into now.

In the future, we may add support for HTTP/2.0 stream format, but for now, we are left with two structural parsers:

* boost::http::reader::request for HTTP/1.0 and HTTP/1.1 request messages.
* boost::http::reader::response for HTTP/1.0 and HTTP/1.1 response messages.

Each structural parser is prepared to receive a continuous stream of messages (i.e. what NodeJS correctly refer to as keep-alive persistent streams). Because the structure of messages is flexible enough to be non-representable in simple non-allocating C++ structures, we don’t decode the whole stream as a single parsing result as this would force allocation. What we do instead is to feed the user with one token at a time and internally we keep a lightweight non-growing state required to decode further tokens.

We use the same token definition for HTTP requests and HTTP responses. The tokens can be either of status (e.g. error or skip), structural (e.g. boost::http::token::code::end\_of\_headers) or data (e.g. boost::http::token::code::field\_name) categories. Only tokens of the data category have an associated value.

Each token is associated with a slice (possibly 0-sized if error token or a token from the structural category) of the byte stream. The process goes as follow:

1. Set the buffer used by the reader.
2. Consume tokens.
   1. Check code() return.
   2. Possibly call value<T>() to extract token value.
   3. Call next().
3. Remove parsed data from the buffer.
   1. You’ll need to keep state of parsed vs unparsed data by calling token\_size().
4. If the address of the unparsed data changes, the reader is invalidated, so to speak. You can restore its valid state by setting the buffer to null or to the new address of the unparsed data.

Enough with abstract info. Take the following part of an HTTP stream:

GET / HTTP/1.1\r\n

Host: www.example.com\r\n

\r\n

This stream can be broken in the following series of tokens (order preserved):

* method.
* request\_target.
* skip.
* version.
* field\_name.
* field\_value.
* skip.
* end\_of\_headers.
* end\_of\_body.
* end\_of\_message.

The parser is required to give you a token\_size() so you can remove parsed data from the stream. However, the parser is not required to give the same series of tokens for the same stream. The strucutral and data tokens will always be emitted the same. However, the parser may choose to merge some status token (e.g. skip) with a data token (e.g. request\_target). Therefore, the following series of tokens would also be possible for the same example given previously:

* method.
* skip.
* request\_target.
* version.
* field\_name.
* skip.
* field\_value.
* end\_of\_headers.
* end\_of\_body.
* end\_of\_message.

This (non-)guarantee is to give freedom to vary the implementation. It’d be absurd to expect different implementations of this interface generating the same result byte by byte. You may expect different algorithms also in future versions.

Another useful feature of this non-guarantee is to make possible to discard skip tokens in the buffer, but merge them if the stream is received in the buffer at once.

Just imagine documenting the guarantees of the token stream if we were to make it predictable. It’d be insane.

However, there is one guarantee that the reader object must provide. It must not discard bytes of data tokens while the token is incomplete. To illustrate this point, let’s go to an example. Given the current token is request\_target, you have the following code.

assert(parser.code() == boost::http::token::code::request\_target);

auto value = parser.value<boost::http::token::request\_target>();

While we traverse the stream, the parser will only *match* tokens. We don’t expect the parser to also *decode* the tokens. The parser will only decode the tokens if necessary to further match the following tokens. And even when the parser decod’em, the intermediary results may be discarded. In other words, *match* and *decode* are separate steps and you can spare CPU time when you don’t need to decode certain elements.

The point is that the token value must be extracted directly from the byte stream and the parser is not allowed to buffer data about the stream (or the decoded values, for that matter). The implication of this rule gives a guarantee about the token order and its relationship to the bytem stream.

You can imagine the stream as having two views. The tokens and the byte streams. The token view spans windows over the byte view.

tokens: | method | skip | request\_target | skip | version

bytes: | GET | <SPC> | / | <SPC> HTTP/1. | 1 <CRLF>

The slice of data associated with a data token can grow larger than the equivalent bytes:

tokens: | method | request\_target | skip | version

bytes: | GET <SPC> | / | <SPC> | HTTP/1.1 <CRLF>

But it cannot shrink smaller than its associated bytes:

tokens: | method | skip | request\_target | skip | version

bytes: | GE | T <SPC> | / | <SPC> HTTP/1.1 | <CRLF>

So you have a token interface easy to inspect and you have a lot of freedom to manage the underlying buffer. Let’s see the boost::http::reader::request parser as used in Tufão:

void HttpServerRequest::onReadyRead()

{

if (priv->timeout)

priv->timer.start(priv->timeout);

priv->buffer += priv->socket.readAll();

priv->parser.set\_buffer(asio::buffer(priv->buffer.data(),

priv->buffer.size()));

Priv::Signals whatEmit(0);

bool is\_upgrade = false;

while(priv->parser.code() != http::token::code::error\_insufficient\_data) {

switch(priv->parser.symbol()) {

case http::token::symbol::error:

priv->socket.close();

return;

case http::token::symbol::skip:

break;

case http::token::symbol::method:

{

clearRequest();

priv->responseOptions = 0;

auto value = priv->parser.value<http::token::method>();

QByteArray method(value.data(), value.size());

priv->method = std::move(method);

}

break;

case http::token::symbol::request\_target:

{

auto value = priv->parser.value<http::token::request\_target>();

QByteArray url(value.data(), value.size());

priv->url = std::move(url);

}

break;

case http::token::symbol::version:

{

auto value = priv->parser.value<http::token::version>();

if (value == 0) {

priv->httpVersion = HttpVersion::HTTP\_1\_0;

priv->responseOptions |= HttpServerResponse::HTTP\_1\_0;

} else {

priv->httpVersion = HttpVersion::HTTP\_1\_1;

priv->responseOptions |= HttpServerResponse::HTTP\_1\_1;

}

}

break;

case http::token::symbol::status\_code:

qFatal("unreachable");

break;

case http::token::symbol::reason\_phrase:

qFatal("unreachable");

break;

case http::token::symbol::field\_name:

case http::token::symbol::trailer\_name:

{

auto value = priv->parser.value<http::token::field\_name>();

priv->lastHeader = QByteArray(value.data(), value.size());

}

break;

case http::token::symbol::field\_value:

{

auto value = priv->parser.value<http::token::field\_value>();

QByteArray header(value.data(), value.size());

priv->headers.insert(priv->lastHeader, std::move(header));

priv->lastHeader.clear();

}

break;

case http::token::symbol::trailer\_value:

{

auto value = priv->parser.value<http::token::trailer\_value>();

QByteArray header(value.data(), value.size());

priv->trailers.insert(priv->lastHeader, std::move(header));

priv->lastHeader.clear();

}

break;

case http::token::symbol::end\_of\_headers:

{

auto it = priv->headers.find("connection");

bool close\_found = false;

bool keep\_alive\_found = false;

for (;it != priv->headers.end();++it) {

auto value = boost::string\_view(it->data(), it->size());

http::header\_value\_any\_of(value, [&](boost::string\_view v) {

if (iequals(v, "close"))

close\_found = true;

if (iequals(v, "keep-alive"))

keep\_alive\_found = true;

if (iequals(v, "upgrade"))

is\_upgrade = true;

return false;

});

if (close\_found)

break;

}

if (!close\_found

&& (priv->httpVersion == HttpVersion::HTTP\_1\_1

|| keep\_alive\_found)) {

priv->responseOptions |= HttpServerResponse::KEEP\_ALIVE;

}

whatEmit = Priv::READY;

}

break;

case http::token::symbol::body\_chunk:

{

auto value = priv->parser.value<http::token::body\_chunk>();

priv->body.append(asio::buffer\_cast<const char\*>(value),

asio::buffer\_size(value));

whatEmit |= Priv::DATA;

}

break;

case http::token::symbol::end\_of\_body:

break;

case http::token::symbol::end\_of\_message:

priv->buffer.remove(0, priv->parser.parsed\_count());

priv->parser.set\_buffer(asio::buffer(priv->buffer.data(),

priv->parser.token\_size()));

whatEmit |= Priv::END;

disconnect(&priv->socket, SIGNAL(readyRead()),

this, SLOT(onReadyRead()));

break;

}

priv->parser.next();

}

priv->buffer.remove(0, priv->parser.parsed\_count());

if (is\_upgrade) {

disconnect(&priv->socket, SIGNAL(readyRead()),

this, SLOT(onReadyRead()));

disconnect(&priv->socket, SIGNAL(disconnected()),

this, SIGNAL(close()));

disconnect(&priv->timer, SIGNAL(timeout()), this, SLOT(onTimeout()));

priv->body.swap(priv->buffer);

emit upgrade();

return;

}

if (whatEmit.testFlag(Priv::READY)) {

whatEmit &= ~Priv::Signals(Priv::READY);

this->disconnect(SIGNAL(data()));

this->disconnect(SIGNAL(end()));

emit ready();

}

if (whatEmit.testFlag(Priv::DATA)) {

whatEmit &= ~Priv::Signals(Priv::DATA);

emit data();

}

if (whatEmit.testFlag(Priv::END)) {

whatEmit &= ~Priv::Signals(Priv::END);

emit end();

return;

}

}

Boost.Http higher level message framework’s socket has a buffer of fixed size and cannot have the luxury of appending data every time. Both high level projects have many fundamental differences.

| **Boost.Http** | **Tufão** |
| --- | --- |
| Boost.Asio active style. | Qt event loop passive style. |
| Boost usage allowed. | It uses this header-only parser lib at Tufão build time and Tufão user will never need Boost again. |
| Message-based framework which allows different backends to be plugged later keeping the same handlers. | Tied to HTTP/1.1 embedded server. |
| Callbacks and completion tokens. It may read more than asked for, but it’ll use read\_state to keep user informed. | Combined with Qt network reactive programming style, it has a strange logic related to event signalling. |
| Proper HTTP upgrade semantics. | Strange HTTP upgrade semantics thanks to the immaturity of following NodeJS design decisions. |
| It normalizes all header keys to lower case. | Case insensitive string classes for the C++ counterpart of the HTTP field names structure. |

These are the main differences that I wanted to note. You can be sure this parser will fit you and it’ll be easy to use. And more importantly, easy to use **right**. [NodeJS parser demands too much HTTP knowledge on the user behalf](https://vinipsmaker.wordpress.com/2016/08/05/boost-http-has-a-new-parser/). And thanks to the NodeJS parser hard to use API, Tufão only was able to support proper HTTP pipelining once it migrated to Boost.Http parser (although Boost.Http managed to do lots of ninja techs to support it under NodeJS parser).

To sum up the data received handler structure, you need:

1. Get the buffer right with parser.set\_buffer(buf).
2. Loop to consume — parser.next() — tokens while http::token::code::error\_insufficient\_data.
   1. Examine token with parser.code().
   2. Maybe handle error.
   3. Extract data with parser.value<T>() if a data token.
3. Remove parsed data.

There are a lot of different HTTP server/client models you can build on top of this framework and the notification style you’re to use is entirely up to you. Most likely, you’ll want to hook some actions when the always-to-be-triggered delimiters category tokens (e.g. boost::http::token::code::end\_of\_headers) are reached.

2.5. Parsing HTTP upgrade

Given you already know the basics, parsing HTTP upgrade is trivial. Because the HTTP parser doesn’t take ownership of the buffer and you pretty much know up until which point the stream was parsed as HTTP.

All you gotta do is consume all the HTTP data (i.e. watch for code::end\_of\_message) and parse the rest of the buffer as the new protocol. Here is the Tufão code to update an HTTP client to WebSocket:

inline bool WebSocketHttpClient::execute(QByteArray &chunk)

{

if (errored)

return false;

parser.set\_buffer(asio::buffer(chunk.data(), chunk.size()));

while(parser.code() != http::token::code::error\_insufficient\_data) {

switch(parser.symbol()) {

case http::token::symbol::error:

errored = true;

return false;

case http::token::symbol::skip:

break;

case http::token::symbol::method:

qFatal("unreachable");

break;

case http::token::symbol::request\_target:

qFatal("unreachable");

break;

case http::token::symbol::version:

if (parser.value<http::token::version>() == 0) {

errored = true;

return false;

}

break;

case http::token::symbol::status\_code:

status\_code = parser.value<http::token::status\_code>();

if (status\_code != 101) {

errored = true;

return false;

}

parser.set\_method("GET");

break;

case http::token::symbol::reason\_phrase:

break;

case http::token::symbol::field\_name:

{

auto value = parser.value<http::token::field\_name>();

lastHeader = QByteArray(value.data(), value.size());

}

break;

case http::token::symbol::field\_value:

{

auto value = parser.value<http::token::field\_value>();

QByteArray header(value.data(), value.size());

headers.insert(lastHeader, std::move(header));

lastHeader.clear();

}

break;

case http::token::symbol::end\_of\_headers:

break;

case http::token::symbol::body\_chunk:

break;

case http::token::symbol::end\_of\_body:

break;

case http::token::symbol::trailer\_name:

break;

case http::token::symbol::trailer\_value:

break;

case http::token::symbol::end\_of\_message:

ready = true;

chunk.remove(0, parser.parsed\_count());

parser.set\_buffer(asio::buffer(chunk.data(),

parser.token\_size()));

break;

}

parser.next();

}

chunk.remove(0, parser.parsed\_count());

if (ready && headers.contains("Upgrade"))

return true;

return false;

}

2.6. Implementing Boost.Beast parser interface

In this tutorial, we’ll show you how to implement Boost.Beast parser interface using Boost.Http parser.

|  |  |
| --- | --- |
| **NOTE** | No prior experience with Boost.Beast is required. |

Boost.Beast parser borrows much of its design from Ryan Dahl’s HTTP parser — the NodeJS parser. This is a design that I do know and used more than once in different projects. However, this is not the only design I know of. I see much of this design as an evolution of the language limitations that you find in the C language.

[I’ve previously written a few complaints about the Ryan Dahl’s HTTP parser](https://vinipsmaker.wordpress.com/2016/08/05/boost-http-has-a-new-parser/). Boost.Beast evolves from this design and takes a few different decisions. We’ll see if, and which, limitations the Boost.Beast parser still carries thanks to this inheritance.

2.6.1. Learning how to use the parser

The parser is represented by a basic\_parser<isRequest, Derived> class. The parser is callback-based just like Ryan Dahl’s HTTP parser and it uses CRTP to avoid the overhead of virtual function calls.

|  |  |
| --- | --- |
| **NOTE** | *DESIGN IMPLICATIONS*  And here we can notice the first difference between Boost.Beast and Boost.Http parsers.  If you design an algorithm to work on the parser object, this algorithm must be a template (and it carries the same drawbacks of a header-only library):  template<class Derived>  void do\_stuff(boost::beast::http::basic\_parser<true, Derived> o);  But if you use Boost.Http parser, this requirement vanishes:  void do\_stuff(boost::http::reader::request o); |

To feed the parser with data, you call basic\_parser::put:

template<

class ConstBufferSequence>

std::size\_t

put(

ConstBufferSequence const& buffers,

error\_code& ec);

The parser will match **and** decode the tokens in the stream.

|  |  |
| --- | --- |
| **NOTE** | *DESIGN IMPLICATIONS*  Match and decoding are always applied together. What does this mean? Not much, given decoding HTTP/1.1 tokens is cheap and most of the time it reduces to return a string\_view of the associated buffer region.  However, the implications of the fundamental model chosen (pull or push) give rise to larger divergences at this point already.  In the Boost.Beast parser, the token is passed to the function callback registered. In the Boost.Http parser, the token is hold by the parser object itself.  It might not seem much difference at the first glance, but consider the problem of composability. If I want to write an algorithm to take a boost::http::reader::request object to read a field\_name token and possibly skip it together with the associated field\_value (maybe that field isn’t of your interest), you only need to wrap the parser object if you’re using the Boost.Http solution [[13](https://boostgsoc14.github.io/boost.http/#_footnotedef_13)]. As for the Boost.Beast parser, you’re left with a long and hackishy inheritance chain that I don’t even want to imagine right now if you were to compose the algorithms.  Just as a concrete illustration of what I meant by the Boost.Http solution:  template<class T /\* = http::reader::request \*/>  void socket\_consumer<T>::parse(asio::buffer buf) { /\* ... \*/ }  http::token::code::value my\_parser\_wrapper::code() const  {  return code\_;  }  void my\_parser\_wrapper::next()  {  // Here I should use a case-insensitive comparison function.  // For simplicity, this step is omitted.  wrapped\_parser.next();  code\_ = wrapped\_parser.code();  if (code\_ == code::field\_name  && (wrapped\_parser.value<token::field\_name>()  == "connection")) {  code\_ = code::skip;  skip\_next\_value = true;  } else if (code\_ == code::field\_value  && skip\_next\_value) {  code\_ = code::skip;  skip\_next\_value = false;  }  }  How would the solution look like using the Boost.Beast parser? Let’s draft something:  template<class Derived>  class my\_parser\_wrapper  : public basic\_parser<true, my\_parser\_wrapper<Derived>>  {  public:  void  on\_field\_impl(field f, string\_view name, string\_view value, error\_code& ec)  {  // Here I should use a case-insensitive comparison function.  // For simplicity, this step is omitted.  if (name == "connection") {  return;  }  static\_cast<Derived&>(\*this).on\_field\_impl(f, name, value, ec);  }  void on\_header\_impl(error\_code& ec)  {  static\_cast<Derived&>(\*this).on\_header\_impl(ec);  }  // ...  };  template<template <typename> class parser>  class custom\_parser  : public parser<custom\_parser>  {  // Here we have our original handler.  // It'd be our `my\_socket\_consumer` from the Boost.Http example.  };  custom\_parser<my\_parser\_wrapper>;  However, this solution is fully flawed. As my\_parser\_wrapper inherits directly from basic\_parser, we cannot inject some my\_parser\_wrapper2 which applies another transformation in the chain.  I can’t emphasize enough that the problem is **not** about adding a “skip-this-set-of-HTTP-headers” function. The problem is about a fundamental building block which can solve more of the user needs. I could keep thinking about the different problems that could happen, but if you do not give a **try** to enter in the *general* problem and insist on a *myopic vision*, you’ll **never** grasp my message (just as an addicted to inductive reasoning will never understand someone who is using deductive reasoning). If all you have is a hammer, everything looks like a nail. We shall see more design implications later on as we continue this chapter. |

As the tokens are found, the user callbacks are called. The function returns the number of parsed bytes.

|  |  |
| --- | --- |
| **NOTE** | *DESIGN IMPLICATIONS*  And as each sentence goes on, it seems that I need to explain more design implications.  What if you want to *reject* messages as soon as one specific token is found? The point here is about avoiding unnecessary computation of parsing elements of a message that would be rejected anyway.  For Boost.Http parser, the control flow is yours to take and…​  ***Do what thou wilt****shall be the whole of the Law.*  *— Aleister Crowley*  A concrete example if you may:  // ...  case code::field\_value:  if (last\_header\_was\_x && is\_good(reader.value<token::field\_value>())) {  // stop the world (e.g. `return` or `throw`)  }  // ...  As for Boost.Beast parser. There **is** an answer, but not with your current limited knowledge of the API. Let’s continue to present Boost.Beast API and come back at this “stop the world” problem later. |

The behaviour usually found in push parsers is to parse the stream until the end of the feeded buffers and then return. This is the NodeJS’s parser approach from which Boost.Beast takes much inspiration. However, Boost.Beast takes a slightly different approach to this problem so it’s possible to parse only one token at a time. The Boost.Beast solution is the eager function:

void

eager(

bool v);

*Normally the parser returns after successfully parsing a structured element (header, chunk header, or chunk body) even if there are octets remaining in the input. This is necessary when attempting to parse the header first, or when the caller wants to inspect information which may be invalidated by subsequent parsing, such as a chunk extension. The eager option controls whether the parser keeps going after parsing structured element if there are octets remaining in the buffer and no error occurs. This option is automatically set or cleared during certain stream operations to improve performance with no change in functionality.*

*The default setting is false.*

*— Vinnie FalcoBoost.Beast documentation*

|  |  |
| --- | --- |
| **NOTE** | *DESIGN IMPLICATIONS*  And now, back at the “stop the world” problem…​  Simply put, Boost.Beast solution is just a hackishy way to implement a pull parser — the parser approach *consciously* chosen by Boost.Http parser.  Alternatively, you can just set the error\_code& ec on the callback implementation to stop parsing, but this wouldn’t solve all the use cases (the reason why eager is provided). |

Continuing this inductive reasoning of “hey! a problem appeared, let’s write yet another function, function\_xyz, to solve use case 777”, a number of other functions are provided. One of them is header\_limit:

void

header\_limit(

std::uint32\_t v);

*This function sets the maximum allowed size of the header including all field name, value, and delimiter characters and also including the CRLF sequences in the serialized input. If the end of the header is not found within the limit of the header size, the error http::header\_limit is returned by http::basic\_parser::put.*

*Setting the limit after any header octets have been parsed results in undefined behavior.*

*— Vinnie FalcoBoost.Beast documentation*

Another function, body\_limit, is provided in the same spirit of header\_limit. What if I have a use case to limit request-target size? Then Boost.Beast author will add function\_xyz2 to use case 778.

|  |  |
| --- | --- |
| **NOTE** | *DESIGN IMPLICATIONS*  What is the Boost.Http solution to this problem 🤔? This is broken into two possible cases.   1. The whole token is in the buffer: In such case you just need to check token\_size. 2. The buffer has been exhausted and no token is there: Here, just check expected\_token.   It’ll work for **any** token (i.e. you don’t need one extra function for each possible token which would just complicate the implementation and inflate the object with a large Settings object of some sort). |

With all this info, the Boost.Beast parser is mostly covered and we can delve into the implementation of such interface.

|  |  |
| --- | --- |
| **NOTE** | *DESIGN IMPLICATIONS*  Now…​ let’s look at something different. Suppose the following scenario:  You have an embedded project and the headers must not be stored (as it’d imply heap memory of complex data structures). You process options with an in situ algorithm out from the headers. In Boost.Http parser, I’m imagining something in these lines:  enum last\_header\_code {  OUT\_OF\_INTEREST\_SET,  FOO,  BAR,  FOOBAR  };  // ...  case code::field\_name:  {  auto v = reader.value<token::field\_name>();  if (iequals(v, "foo")) {  last\_header = FOO;  } else if (iequals(v, "bar")) {  last\_header = BAR;  } else if (iequals(v, "foobar")) {  last\_header = FOOBAR;  } else {  last\_header = OUT\_OF\_INTEREST\_SET;  }  break;  }  case code::field\_value:  if (last\_header == FOO) {  foo = process\_foo(reader.value<token::field\_value>());  } else if (last\_header == BAR) {  bar = process\_bar(reader.value<token::field\_value>());  } else if (last\_header == FOOBAR) {  foobar = process\_foobar(reader.value<token::field\_value>());  }  break;  // ...  Boost.Beast solution is not hard to imagine too:  // ...  void on\_field\_impl(field, string\_view name, string\_view value, error\_code& ec)  {  if (iequals(name, "foo")) {  foo = process\_foo(value);  } else if (iequals(name, "bar")) {  bar = process\_bar(value);  } else if (iequals(name, "foobar")) {  foobar = process\_foobar(value);  }  }  // ...  So…​ what does each design *implies*? As Boost.Beast parser always parse field name + field value *together*, if both fields sum up more than the buffer size, you’re out of luck. Both tokens must fit in the buffer together.  Just as an exercise, let’s pursue the inductive reasoning applied to this problem. We *could* split the Boost.Beast’s on\_field\_impl callback into two:  void  on\_field\_name\_impl(  field f,  string\_view name,  error\_code& ec);  void  on\_field\_value\_impl(  string\_view value,  error\_code& ec);  But then we create another problem:  *[…​] it is the responsibility of the derived class to copy any information it needs before returning from the callback […​]*  *— Vinnie FalcoBoost.Beast documentation*  If you don’t see a problem already, let me unveil it for you. Now, most of the uses of the parser, which want to store the HTTP headers in some sort of std::multimap structure will have to perform one extra allocation:  void on\_field\_name\_impl(field, string\_view name, error\_code& ec)  {  last\_header = to\_string(name);  }  Under the push parser model, these two cases are irreconcilable. Boost.Beast opts to solve the most common problem and this was a good design choice (let’s give credit where credit is due).  However, Boost.Http parser is a good choice in *any* of these two cases. It only feeds one token at a time. And as Boost.Http message framework demonstrate, [we can use the first bytes of the buffer to store the HTTP field name](https://github.com/BoostGSoC14/boost.http/blob/c7b3e11884420b55568222d237e9c429ec24b811/include/boost/http/socket-inl.hpp#L630).  And just to present a more readable alternative, you could play with copies of the reader object made in the stack of the my\_socket\_consumer::on\_socket\_callback function. This way, you have a point in time and you can make the parser “go back”. The copies are cheap because the reader object is just an integer-based state machine with a few indexes. The *idea* behind this solution is to mirror current Boost.Beast behaviour — field name and field value are always kept together in the buffer.  **Remember**…​ principles. I can attack other specific cases. As an exercise, try to find a few yourself. |

2.6.2. Implementing the Boost.Beast interface

|  |  |
| --- | --- |
| **NOTE** | As we previously seen, there are several functions in Boost.Beast parser that are just boilerplate inherited (e.g. eager) thanks to the choice of the wrong fundamental model (i.e. pull vs push).  We’ll skip some of this boilerplate as it is not of our interest. Our purpose with this tutorial was to show design implications derived from the choices of the *fundamental* models. |

template<bool isRequest, class Derived>

class basic\_parser;

// This template specialization is wrong, but is kept for simplification

// purposes.

template<bool isRequest, class Derived>

class basic\_parser<true, Derived>

{

public:

template<

class ConstBufferSequence>

std::size\_t

put(

ConstBufferSequence const& buffers,

error\_code& ec)

{

// WARNING: the real implementation will have more trouble because of

// the `ConstBufferSequence` concept, but for the reason of simplicity,

// we don't show the real code here.

reader.set\_buffer(buffers);

error\_code ec;

while (reader.code() != code::error\_insufficient\_data) {

switch (reader.code()) {

case code::skip:

break;

case code::method:

method = reader.value<token::method>;

break;

case code::request\_target:

target = reader.value<token::request\_target>;

break;

case code::version:

static\_cast<Derived&>(\*this)

.on\_request\_impl(/\*the enum code\*/, method, target,

reader.value<token::version>(), ec);

if (ec) {

// TODO: extra code to enter in error state

return reader.parsed\_count();

}

break;

// ...

case code::end\_of\_headers:

static\_cast<Derived&>(\*this).on\_header\_impl(ec);

if (ec) {

// TODO: extra code to enter in error state

return reader.parsed\_count();

}

break;

// ...

}

reader.next();

}

return reader.parsed\_count();

}

private:

boost::http::reader::request reader;

// It's possible and easy to create an implementation that doesn't allocate

// memory. Just keep a copy of `reader` within the `put` function body and

// you can go back. As `reader` is just an integer-based state machine with

// a few indexes, the copy is cheap. I'm sorry I don't have the time to code

// the demonstration right now.

std::string method;

std::string target;

};

A final note I want to add is that I plan more improvements to the parser. Just as Boost.Beast parser is an evolution of the wrong model chosen for the problem, my parser still has room to *evolve*. But from my judgment, this parser already is better than Boost.Beast parser can ever be (i.e. the problems I presented here are unfixable in Boost.Beast design…​ not to mention that Boost.Beast parser has almost the double amount of member-functions to solve *the same* problem [[14](https://boostgsoc14.github.io/boost.http/#_footnotedef_14)]).

3. Design choices

3.1. FAQ

1. *Why build on top of Boost.Asio?*

One of the requirements of Boost.Http is scalable performance. Scalable performance requires an asynchronous design. Asio is expected to become the future C++ standard for sockets. So, Asio it is.

Also, reuse existing std or Boost components is nice and Asio is the Boost solution for asynchronous socket I/O.

1. *Why C++11?*

Asynchronous code can be largely simplified with language support for lambdas. To speed up the development (and decrease bugs), C++11 was chosen, but interface-wise, the only real C++11 feature is enum classes, which can be emulated in C++98 easily.

C++98 support might be added later. The priority is to prove the core set of abstractions is correct. Then I can move on the task to fatten the library. The proof comes in the act of passing the Boost review process.

To be fair, I also depend on the C++11 definition of multimap, but Boost containers can do the job easily.

1. *Have you considered contribute to*project X/Y*?*

[Yes, I have](https://github.com/vinipsmaker/gsoc2014-boost/blob/master/other_frameworks.md).

But current Boost.Http is not like what I was expecting when I started to writing it. Boost.Http is better. The gap between Boost.Http and other projects became even larger. My previous research won’t mention every difference.

Boost.Http supports pipelining. Pion has separate functions for chunking. Boost.Http is designed for multiple backends. POCO, QtHttp and Casablanca aren’t build on top of Asio. Pion and cpp-netlib will use their own thread-pool, instead adhering to Asio threading model.

1. *Why is it only server-side?*

Server-side and client-side are of interest to different applications. Initially, the focus was to provide a library just to server-side, but with the time spent on research and development, it became apparent that many of the proposed abstractions are also useful for client-side.

After this fact, a lot of caution was devoted to design the interface to retain the usefulness in client-side, where it makes sense. But this is not enough. A lot of time was spent on research just to get the server-side right and I expect that much time (or more) to also get the client-side right.

Before any serious effort is spent on client-side, I want to focus on server-side, where the application load may be way higher and C++ may be way more desired. And just as the server-side interface development was driven by a strict set of guidelines (multiple backends, modularity with specific use cases…​), we need to define what we want to achieve with the client-side abstraction. What kind of usage will be appropriate with such design.

1. *Why isn’t a router available?*

Advocates of tree-based routers tend to ignore the middleware-based approach and the other way around is also true. It happens that some even only know one way and don’t even stop to consider that their approach isn’t appropriate for every project. This subject will affect the life of the users **a lot** and can be rather polemic.

I just provide the building blocks and you can create the router any way you want. Actually, I intend to implement them later, because implementing them now will just distract the attention of the reviewers and it’d be a waste of time if the review proves the core set of abstractions is wrong.

Most of the designs I see propose dynamic routers, where you can change the routing rules at runtime, but this feature is rarely needed. Wouldn’t be wonderful if you could use great syntax sugars to declare routers that receive as much optimization as possible at compile-time? Wouldn’t be wonderful to use nested routers? Wouldn’t be wonderful if you could collaborate the tree-based and middleware-based approach very easily? Maybe even some kind of collaboration between the statically declared routers and dynamic routers? I hope this will be rather polemic and will require a lot of iterations to get it right, maybe with a mini-review for acceptance.

1. *How robust is this parser?*

It doesn’t try to parse URLs at all. It’ll ensure that only valid characters (according to HTTP request target BNF rule) are present, but invalid sequences are accepted.

This parser is a little (but not too much) more liberal in what accepts and it’ll accept invalid sequences for rarely used elements that don’t impact upper layers of the application. The reason to accept such non-conformant sequences is a simpler algorithm that can be more performant (e.g. we only check for invalid chars, but not invalid sequences). Therefore, it’s advised to reframe the message if you intend to forward it to some other participant. Not doing so, might be a security issue if the participant you are forwarding your message to is know to show improper behaviour when parsing invalid streams. There are several references within the RFC7230 where similar decisions are suggested (e.g. if you receive redundant Content-Length header field, you must merge it into one before forwarding or reject the message as a whole).

1. *Why not make a parser using Boost.Spirit?*

Boost.Spirit needs backtracking to implement the OR operator. Boost.Spirit can’t build a state machine which would allow you to continue parsing from the suspended point/byte. Thanks to these characteristics, it can’t be used in our HTTP parser. Also, we don’t see much benefit in pursuing this effort.

1. *What is the recommended buffer size?*

A buffer of size 7990 is recommended (suggested request line of 8000 by section 3.1.1 of RFC7230 minus spaces, minus http version information minus the minimum size of the other token in the request line). However, the real suggested buffer size should be how long names you expect to have on your own servers.

1. *What are the differences between reader::request and reader::response?*
   * response has the void set\_method(view\_type method) member-function.

|  |  |
| --- | --- |
| **WARNING** | This member-function **MUST** be called for each HTTP response message being parsed in the stream. |

* + response has the void puteof() member-function.
  + code() member function return value has different guarantees in each class.
  + template<class T> typename T::type value() const member function accepts different input template arguments in each class.

3.2. Design choices

To convince you about the solution, I’ll start the text highlighting some problems and requirements. If you understand the problem, you’ll understand why the solution was proposed like that.

One of the wanted features for this library since the very beginning was to make it possible to make a small change in code to expose the HTTP service through a different communication mechanism. Like, change from embedded HTTP server to FastCGI-exposed server.

There are several libraries who will provide some object you can use to consume HTTP traffic from the world and will act as a request-reply door to create web applications. Libraries who will expose a request object you can use to consume TCP traffic and will export url and headers properties. The problem with this approach is the coupling between HTTP messages and HTTP communication channels.

In Boost.Http, HTTP messages and HTTP communication channels are decoupled, so it is easier to replace the communication channel later. You could easily use an unprotected embedded HTTP server on development environment to tests and replace it in favor of a full-blow solution during production.

Boost.Http defines some type requirements to abstract communication channels and provide some polymorphic adapters who will type erase them. The abstraction was specified carefully to allow robust applications. Your application will not hang trying to live stream a video because the request was done from an HTTP/1.0 client. Also, your handler won’t know what HTTP version (if any) the HTTP request was made with.

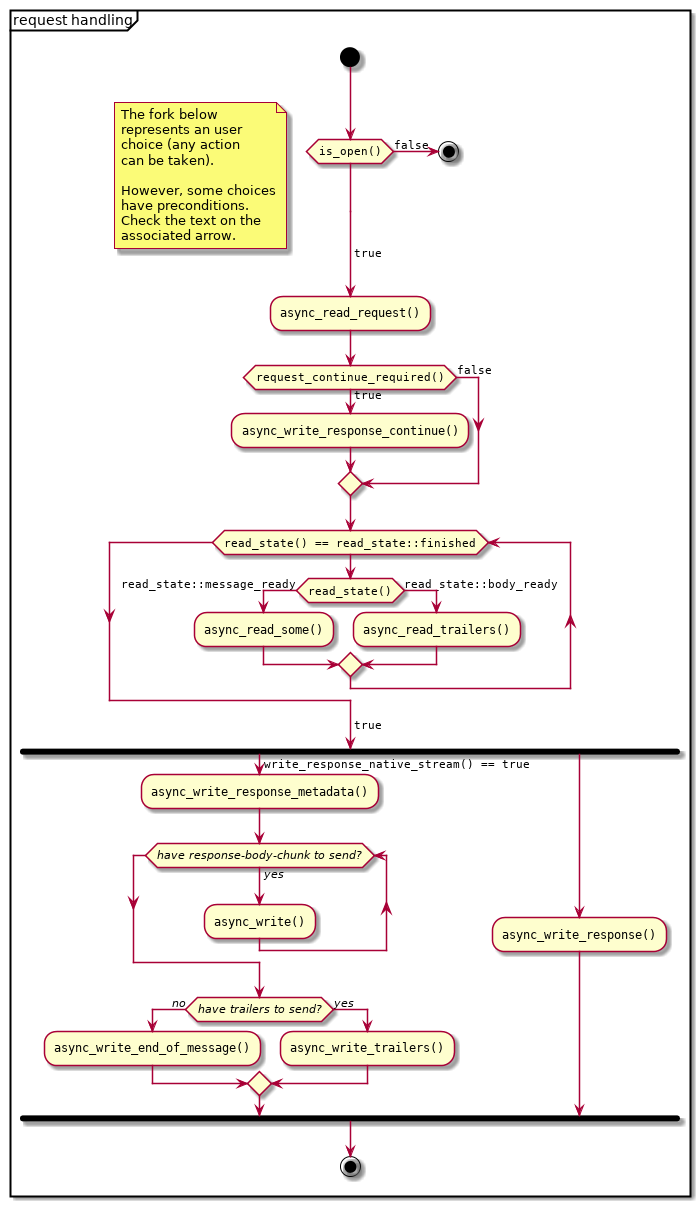
Also among the wanted features was to retain the usefulness of the library whether you’re using it to power an application intended to run from an low-end embedded device or an application intended to run on a cluster with plenty of resources to be made use of. An embdeded device may not have the luxury to host a pool or a cache layer, but a cluster may even demand these layers to properly handle thousands of requests every second. With this use case in mind, modularity was achieved.

The plan to finish such ambitious project was “to expose an HTTP abstraction able to make use of the HTTP power (chunking/streaming, pipelining, upgrade for supporting channels and multiplexing for supporting channels), at the same time that a complete separation of communication channels and HTTP messages is achieved”.

With the separation of HTTP messages and HTTP communication channels, alongside the use of an active model (you ask by the next request instead providing a handler and waiting for them), several of the requirements became very easy to fulfill, such as HTTP pipelining, custom memory allocation, buffers, cache layers and pools of objects.

With such very generalized abstractions, you may be worried about the need to type too much to get something done. This is being solved by providing higher level flexible abstractions, such as the file server you can already find.

3.2.1. The what



The above image shows an overview of how to build HTTP servers using Boost.Http.

Let’s assume you’re going to use the provided http::socket, suitable for embedding HTTP servers in your application.

You must instantiate a TCP socket acceptor and accept new connections. You should handle all connections concurrently.

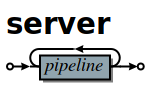
Each connection you handle is a pipeline of request-reply pair. To receive the full request, your action may be required at several points. First, you should receive the request metadata, then it may be necessary to issue a 100-continue response. Then you need to issue reads for the body and the trailers until the whole request has been received.

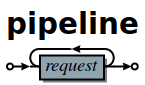
You’re finally able to send the reply. You have two options, atomic messages or chunked messages. Chunked messages are not always available and must check if they can be used for each request you receive, using write\_response\_native\_stream().

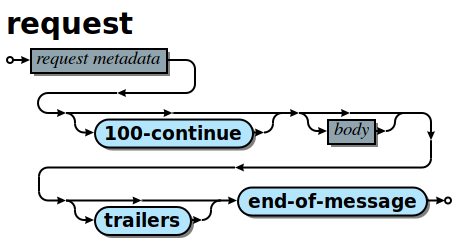
If you spread the handling logic among several functions, a good approach would be to always share the triplet <communication channel, request message, response message> around.

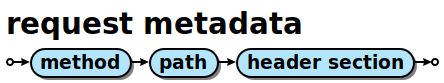
Still missing is URL parsing and request routing, so you must do this yourself, possibly managing pools of message and socket objects.

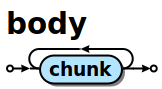
This system allows you to implement powerful schedulers doing fair share of resources over different IPs, whether the requests originate from HTTP or HTTPS, using all cores of your CPU and deferring new work when the work load is too high. You should be able to do all fine-grained tuning you need and also easily create higher level that are suitable for your application. Not only that, this library could become an interoperability layer for all higher-level that web application developers create.











Also, if you pay attention, you’ll realize that this proposal just expose HTTP with a message oriented abstraction. All procedures in the diagram are related to HTTP events and actions. And this is a modern API and you can use pretty much every modern HTTP feature (persistent streams & HTTP pipelining, chunked entities, 100-continue status, …​). And you won’t handle any parsing or low-level detail at all. It’s abstracted enough to allow alternative backends.

However, this can easily become a callback hell, and futures wouldn’t help much, given the need to use while-constructs. If you use coroutines, there is hope your code will be readable. Boost.Http follows Asio extensible asynchronous model and you’re free to use callbacks, futures, coroutines or others.

3.2.2. ASIO familiarity

This library may be very pleasant to use for any ASIO-centered mind.

* Completion tokens received as the last argument for aync functions.
* Async operations have the async\_ prefix.
* User control the bufferring mechanism, passing the opaque asio::buffer type.
* User provides *output* arguments as references and they’ll be “filled” by the time the operation completes.
* Memory management is left for the user.
* An active model is presented.
* Similar nomenclature.

*The ASIO way* saved us from many problems that otherwise would force us to propose solutions to already know problems such as:

* Object pools.
* Deferring acceptance to later on high load scenarios.
* HTTP pipelining problems.
* Partially filling response objects from different layers of abstractions.
* A wrapping/wrapped socket can take care of tasks such as synchronization/queueing and timeout.

3.2.3. The mysterious/weird/news API

One of the maybe surprising things to start with is the use of highly structured objects as opposed to things like opaque buffers. You pass a message object to the initiating function and you’ll have a fully decomposed object with an URL, a method and even an associative container for the headers!!!

If you do have special memory requirements for the messages, you’re free to implementing an alternative container, as long as it fulfills the documented Message concept. Connections channels and HTTP messages are **not** coupled together. You can reuse these pieces in many many different contexts.

The uncoupled architecture is more general and it is the default mode, but let’s say you work at a more constrained environment where memory copying is banned, for instance. You could provide your HTTP backend (e.g. a non-copying embedded server) tied to your specific HTTP message type implementing our ideas and you still may benefit from this libray. This library provides some HTTP algorithms and some HTTP handlers (e.g. file server) and these abstractions will save some time from you.

Another difference in this library is the presence of an associated state for reading and writing messages. I believe this abstraction can be extended to also support very simple HTTP clients. To avoid confusion, if some member-function cannot be used for both modes (clients and servers), it’ll have one of the following prefixes:

* async\_read\_request
* async\_read\_response
* async\_write\_request
* async\_write\_response

We gave special attention to read\_state and write\_state to make sure it’ll also be usable for **simple** and asynchronous HTTP clients.

3.2.4. The why

Boost.Http provides an HTTP socket, which can be used to manage a pipeline of HTTP messages (i.e. an HTTP request or an HTTP reply). HTTP is stateless and each message coming from the same socket is independent. The HTTP socket from Boost.Http is a concept and specific implementations from this concept may provide more guarantees about the communication properties. The reasons to provide few guarantees are (#1) because we want a common denominator from which we can provide implementation for multiple communication channels and (#2) because implementation details are usually not required for the application, which is only interested in a high-level abstraction. The provided boost::http::basic\_socket implementation will handle actual HTTP traffic from TCP sockets and you can use it to handle HTTP/1.0 and HTTP/1.1 traffic from TCP and SSL sockets.

read\_state() and write\_state() are used to inspect the current state of interaction and react appropriately. There are rules regarding when the socket can mutate and change its states. Once you request the socket to read a new HTTP request, you’ll be notified as soon as the request metadata (request line and HTTP headers) are ready, then you can progressively download the body and react appropriately. This idea is very useful to improve communication between the library authors and application authors and also helps to create some tests.

You’ll have to inspect the socket to know whether the current message-exchange requires 100-continue, allows chunked entities (streaming response) and alike. There is like two kind of replies. With atomic replies, you write the whole message at once. With chunked message, you compose a message spreading its construction among several API calls. You may want to use chunked messages when you don’t know the whole body in advance (e.g. reading a file, video live stream…​), but chunked messages can only be used in certain message exchanges. The reason behind providing two kind of replies is to properly support a wider range of HTTP communication channels.

You create one HTTP socket for each HTTP client and should handle them all concurrently. In case you’re using the embeddable HTTP server backend, you must use an acceptor to initialize the basic\_sockets' next\_layer() and then consume them. basic\_socket templatize the underlying internal socket, so you can use SSL, queue wrapping socket (to work around Asio’s composed operations) and so on. The intention of Boost.Http is not only to generalize over data structures and HTTP backends, but about any place where it may be helpful.

The choice to represent the HTTP messages in separate objects and the whole combination of this design ease supports for HTTP pipelining a lot. In passive styles, a request is generated and generated and you must act on them. In this active style, you explicitly request the next message, handle it and then request another one. In this scenario, two unrelated messages won’t be mixed up, because you won’t see the next message while you don’t handle the current one. The read and write states gives a mean to communicate how to use the API and how to detect some logical errors in the application.

The choice to hide details from the HTTP connection (HTTP version, socket object…​) was done to properly support multiple backends. The ability to query certain properties from the underlying communication channel is necessary to achieve reliability under this model. A lot of responsibilies and expected behaviour is documented on the type requirements for ServerSocket objects.

A C++11 multimap is used to represent HTTP headers because that’s what HTTP headers conceptually are. HTTP spec specifies you must handle HTTP header elements with equivalent keys as if there was a single header where the values are joined with commas. Some old headers don’t work with this approach and their values, when multiple elements with equivalent keys are present, must be stored separately. The order matters, just as the C++11 definition of multimap.

Runtime-based polymorphic behaviour isn’t used by default, because not all projects are willing to pay for this price. Well defined type requirements are provided and some polymorphic adaptors will convert models of these type requirements to classes inheriting a single specific abstract base class.

Member-functions as opposed to member-variables are used in HTTP messages, because some setup (e.g. a proxy who doesn’t want to reformat the messages) may want to move the HTTP parser to the HTTP message object. I want to allow a library who will beat C servers in every aspect.

As per [RFC 7230](https://tools.ietf.org/html/rfc7230#section-3.2.2), “a server MUST NOT apply a request to the target resource until the entire request header section is received, since later header fields might include conditionals, authentication credentials, or deliberately misleading duplicate header fields that would impact request processing”, so we define an interface who will only expose a message once the complete header section is ready. The message body can be progressively received later. The API also unifies HTTP messages and HTTP chunking.

URL-decomposed objects aren’t used because all an HTTP backend needs is some string-like container to push bytes. This container can implement an in-place URL parsing algorithm and it is all solved. The generic HTTP backends you find in Boost.Http won’t care about the url concrete type and you don’t need to expect any barrier from this side.

We do not use the message itself as a buffer object while we’re parsing the connection stream. We require a separate buffer to be able to properly handle HTTP pipelining (and futurely multiplexing in HTTP/2.0).

3.2.5. The when

I couldn’t resist the temptation of adding a “*when*” named section after I already had written a “*what*” and a “*why*” section.

Just too much research time went into this proposal. Really, a lot of time. I developed some broken HTTP projects some years ago, learned a lot of design with really different approaches (PHP, Django, Node.js) trying to solve this problem, developed my own serious project ([Tufão](https://github.com/vinipsmaker/tufao)) and continued to study and research a lot (the HTTP spec resurrection project, or [RFC 7230](https://tools.ietf.org/html/rfc7230#section-3.2.2), helped **a lot**). I’ve gathered info around where interoperability may be a problem if API doesn’t help and what features will be desired, sooner or later, by users, among other data. I’ve done real effort to gather feedback from C++ programmers for quite a while already.

A special thanks to Bjørn Reese for mentoring me on Asio quirks and API general design, the feedback which changed the proposal the most. Also a special thanks to any friend who helped to maintain my mind at a happy state.

3.3. Roadmap

* C++98.
* Client-side HTTP.
* HTTP/2.0.
* Request-router.
* Forms and file uploads.
* Cookies and sessions (RFC 6265).
* WebSocket.
* Alternative backends.
* Increase test coverage a lot.
* Benchmarks.
* Compress replies.
* WebDAV (it will depend on Boost.XML, which doesn’t exist yet).
* World domination.

*Parser layer*

* Parsers combinators.
* Incremental message generator.
* Iterator adaptors.

4. Reference

All declarations from this library resides within the boost::http namespace. For brevity, this prefix is **not** repeated on the documentation.

4.1. Summary

4.1.1. Classes

* [headers](https://boostgsoc14.github.io/boost.http/#headers)
* [request](https://boostgsoc14.github.io/boost.http/#request)
* [response](https://boostgsoc14.github.io/boost.http/#response)
* [socket](https://boostgsoc14.github.io/boost.http/#socket)
* [buffered\_socket](https://boostgsoc14.github.io/boost.http/#buffered_socket)
* [poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#poly_socket_base)
* [poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#poly_server_socket)
* [poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#poly_client_socket)
* [poly\_socket](https://boostgsoc14.github.io/boost.http/#poly_socket)
* Tokens
  + [token::skip](https://boostgsoc14.github.io/boost.http/#token_skip)
  + [token::field\_name](https://boostgsoc14.github.io/boost.http/#token_field_name)
  + [token::field\_value](https://boostgsoc14.github.io/boost.http/#token_field_value)
  + [token::body\_chunk](https://boostgsoc14.github.io/boost.http/#token_body_chunk)
  + [token::end\_of\_headers](https://boostgsoc14.github.io/boost.http/#token_end_of_headers)
  + [token::end\_of\_body](https://boostgsoc14.github.io/boost.http/#token_end_of_body)
  + [token::trailer\_name](https://boostgsoc14.github.io/boost.http/#token_trailer_name)
  + [token::trailer\_value](https://boostgsoc14.github.io/boost.http/#token_trailer_value)
  + [token::end\_of\_message](https://boostgsoc14.github.io/boost.http/#token_end_of_message)
  + [token::method](https://boostgsoc14.github.io/boost.http/#token_method)
  + [token::request\_target](https://boostgsoc14.github.io/boost.http/#token_request_target)
  + [token::version](https://boostgsoc14.github.io/boost.http/#token_version)
  + [token::status\_code](https://boostgsoc14.github.io/boost.http/#token_status_code)
  + [token::reason\_phrase](https://boostgsoc14.github.io/boost.http/#token_reason_phrase)
* Structural parsers
  + [reader::request](https://boostgsoc14.github.io/boost.http/#reader_request)
  + [reader::response](https://boostgsoc14.github.io/boost.http/#reader_response)

4.1.2. Class Templates

* [basic\_request](https://boostgsoc14.github.io/boost.http/#basic_request)
* [basic\_response](https://boostgsoc14.github.io/boost.http/#basic_response)
* [basic\_socket](https://boostgsoc14.github.io/boost.http/#basic_socket)
* [basic\_buffered\_socket](https://boostgsoc14.github.io/boost.http/#basic_buffered_socket)
* [request\_response\_wrapper](https://boostgsoc14.github.io/boost.http/#request_response_wrapper)
* [basic\_poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#basic_poly_socket_base)
* [basic\_poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_server_socket)
* [basic\_poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_client_socket)
* [server\_socket\_adaptor](https://boostgsoc14.github.io/boost.http/#server_socket_adaptor)
* [client\_socket\_adaptor](https://boostgsoc14.github.io/boost.http/#client_socket_adaptor)
* [socket\_adaptor](https://boostgsoc14.github.io/boost.http/#socket_adaptor)
* [basic\_poly\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_socket)
* [is\_message](https://boostgsoc14.github.io/boost.http/#is_message)
* [is\_request\_message](https://boostgsoc14.github.io/boost.http/#is_request_message)
* [is\_response\_message](https://boostgsoc14.github.io/boost.http/#is_response_message)
* [is\_socket](https://boostgsoc14.github.io/boost.http/#is_socket)
* [is\_server\_socket](https://boostgsoc14.github.io/boost.http/#is_server_socket)
* [is\_client\_socket](https://boostgsoc14.github.io/boost.http/#is_client_socket)
* [basic\_router](https://boostgsoc14.github.io/boost.http/#basic_router)
* [regex\_router](https://boostgsoc14.github.io/boost.http/#regex_router)
* Content parsers
  + [syntax::chunk\_size](https://boostgsoc14.github.io/boost.http/#syntax_chunk_size)
  + [syntax::content\_length](https://boostgsoc14.github.io/boost.http/#syntax_content_length)
  + [syntax::strict\_crlf](https://boostgsoc14.github.io/boost.http/#syntax_strict_crlf)
  + [syntax::liberal\_crlf](https://boostgsoc14.github.io/boost.http/#syntax_liberal_crlf)
  + [syntax::field\_name](https://boostgsoc14.github.io/boost.http/#syntax_field_name)
  + [syntax::left\_trimmed\_field\_value](https://boostgsoc14.github.io/boost.http/#syntax_left_trimmed_field_value)
  + [syntax::ows](https://boostgsoc14.github.io/boost.http/#syntax_ows)
  + [syntax::reason\_phrase](https://boostgsoc14.github.io/boost.http/#syntax_reason_phrase)
  + [syntax::status\_code](https://boostgsoc14.github.io/boost.http/#syntax_status_code)

4.1.3. Free Functions

* Header processing
  + [header\_to\_ptime](https://boostgsoc14.github.io/boost.http/#header_to_ptime)
  + [to\_http\_date](https://boostgsoc14.github.io/boost.http/#to_http_date)
  + [header\_value\_all\_of](https://boostgsoc14.github.io/boost.http/#header_value_all_of)
  + [header\_value\_any\_of](https://boostgsoc14.github.io/boost.http/#header_value_any_of)
  + [header\_value\_none\_of](https://boostgsoc14.github.io/boost.http/#header_value_none_of)
  + [header\_value\_for\_each](https://boostgsoc14.github.io/boost.http/#header_value_for_each)
  + [etag\_match\_strong](https://boostgsoc14.github.io/boost.http/#etag_match_strong)
  + [etag\_match\_weak](https://boostgsoc14.github.io/boost.http/#etag_match_weak)
* Channel querying
  + [request\_continue\_required](https://boostgsoc14.github.io/boost.http/#request_continue_required)
  + [request\_upgrade\_desired](https://boostgsoc14.github.io/boost.http/#request_upgrade_desired)
* File server
  + [async\_response\_transmit\_file](https://boostgsoc14.github.io/boost.http/#async_response_transmit_file)
  + [async\_response\_transmit\_dir](https://boostgsoc14.github.io/boost.http/#async_response_transmit_dir)

4.1.4. Enumerations

* [read\_state](https://boostgsoc14.github.io/boost.http/#read_state)
* [write\_state](https://boostgsoc14.github.io/boost.http/#write_state)
* [status\_code](https://boostgsoc14.github.io/boost.http/#status_code)
* [token::code::value](https://boostgsoc14.github.io/boost.http/#token_code_value)
* [token::symbol::value](https://boostgsoc14.github.io/boost.http/#token_symbol_value)
* [token::category::value](https://boostgsoc14.github.io/boost.http/#token_category_value)

4.1.5. Error Codes

* [http\_errc](https://boostgsoc14.github.io/boost.http/#http_errc)
* [file\_server\_errc](https://boostgsoc14.github.io/boost.http/#file_server_errc)

4.1.6. Type Requirements

* [Message](https://boostgsoc14.github.io/boost.http/#message_concept)
* [Request](https://boostgsoc14.github.io/boost.http/#request_concept)
* [Response](https://boostgsoc14.github.io/boost.http/#response_concept)
* [Socket](https://boostgsoc14.github.io/boost.http/#socket_concept)
* [ServerSocket](https://boostgsoc14.github.io/boost.http/#server_socket_concept)
* [ClientSocket](https://boostgsoc14.github.io/boost.http/#client_socket_concept)

4.1.7. Headers

* [<boost/http/algorithm.hpp>](https://boostgsoc14.github.io/boost.http/#algorithm_header)
* [<boost/http/algorithm/header.hpp>](https://boostgsoc14.github.io/boost.http/#header_header)
* [<boost/http/algorithm/query.hpp>](https://boostgsoc14.github.io/boost.http/#query_header)
* [<boost/http/file\_server.hpp>](https://boostgsoc14.github.io/boost.http/#file_server_header)
* [<boost/http/headers.hpp>](https://boostgsoc14.github.io/boost.http/#headers_header)
* [<boost/http/http\_category.hpp>](https://boostgsoc14.github.io/boost.http/#http_category_header)
* [<boost/http/http\_errc.hpp>](https://boostgsoc14.github.io/boost.http/#http_errc_header)
* [<boost/http/request.hpp>](https://boostgsoc14.github.io/boost.http/#request_header)
* [<boost/http/response.hpp>](https://boostgsoc14.github.io/boost.http/#response_header)
* [<boost/http/request\_response\_wrapper.hpp>](https://boostgsoc14.github.io/boost.http/#request_response_wrapper_header)
* [<boost/http/poly\_server\_socket.hpp>](https://boostgsoc14.github.io/boost.http/#poly_server_socket_header)
* [<boost/http/poly\_client\_socket.hpp>](https://boostgsoc14.github.io/boost.http/#poly_client_socket_header)
* [<boost/http/poly\_socket\_base.hpp>](https://boostgsoc14.github.io/boost.http/#poly_socket_base_header)
* [<boost/http/poly\_socket.hpp>](https://boostgsoc14.github.io/boost.http/#poly_socket_header)
* [<boost/http/read\_state.hpp>](https://boostgsoc14.github.io/boost.http/#read_state_header)
* [<boost/http/server\_socket\_adaptor.hpp>](https://boostgsoc14.github.io/boost.http/#server_socket_adaptor_header)
* [<boost/http/client\_socket\_adaptor.hpp>](https://boostgsoc14.github.io/boost.http/#client_socket_adaptor_header)
* [<boost/http/socket\_adaptor.hpp>](https://boostgsoc14.github.io/boost.http/#socket_adaptor_header)
* [<boost/http/socket.hpp>](https://boostgsoc14.github.io/boost.http/#socket_header)
* [<boost/http/buffered\_socket.hpp>](https://boostgsoc14.github.io/boost.http/#buffered_socket_header)
* [<boost/http/status\_code.hpp>](https://boostgsoc14.github.io/boost.http/#status_code_header)
* [<boost/http/write\_state.hpp>](https://boostgsoc14.github.io/boost.http/#write_state_header)
* [<boost/http/traits.hpp>](https://boostgsoc14.github.io/boost.http/#traits_header)
* [<boost/http/basic\_router.hpp>](https://boostgsoc14.github.io/boost.http/#basic_router_header)
* [<boost/http/regex\_router.hpp>](https://boostgsoc14.github.io/boost.http/#regex_router_header)
* [<boost/http/token.hpp>](https://boostgsoc14.github.io/boost.http/#token_header)
* [<boost/http/reader/request.hpp>](https://boostgsoc14.github.io/boost.http/#reader_request_header)
* [<boost/http/reader/response.hpp>](https://boostgsoc14.github.io/boost.http/#reader_response_header)
* [<boost/http/syntax/chunk\_size.hpp>](https://boostgsoc14.github.io/boost.http/#syntax_chunk_size_header)
* [<boost/http/syntax/content\_length.hpp>](https://boostgsoc14.github.io/boost.http/#syntax_content_length_header)
* [<boost/http/syntax/crlf.hpp>](https://boostgsoc14.github.io/boost.http/#syntax_crlf_header)
* [<boost/http/syntax/field\_name.hpp>](https://boostgsoc14.github.io/boost.http/#syntax_field_name_header)
* [<boost/http/syntax/field\_value.hpp>](https://boostgsoc14.github.io/boost.http/#syntax_field_value_header)
* [<boost/http/syntax/ows.hpp>](https://boostgsoc14.github.io/boost.http/#syntax_ows_header)
* [<boost/http/syntax/reason\_phrase.hpp>](https://boostgsoc14.github.io/boost.http/#syntax_reason_phrase_header)
* [<boost/http/syntax/status\_code.hpp>](https://boostgsoc14.github.io/boost.http/#syntax_status_code_header)

4.1.8. Macros

BOOST\_HTTP\_SOCKET\_DEFAULT\_BUFFER\_SIZE

This macro defines the default buffer size for [basic\_buffered\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_buffered_socket). It’s safe to override this value (per using class or globally) and should be done before including the file [<boost/http/buffered\_socket.hpp>](https://boostgsoc14.github.io/boost.http/#buffered_socket_header). The default provided value (i.e. the non-overriden version) is unspecified (e.g. can change among versions and platforms).

BOOST\_HTTP\_UPGRADE\_HEAD\_DISABLE\_CHECK

Define this macro if you want to disable the check in basic\_socket::upgrade\_head. The check is there to ensure that you only uses this function in HTTP client mode. It’ll throw an exception to notify the violation.

BOOST\_HTTP\_SOCKET\_DEFAULT\_BODY\_COPY\_THRESHOLD

This macro defines the default value for [basic\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_socket)'s Settings::body\_copy\_threshold. It represents the maximum amount of bytes [basic\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_socket) will copy from message bodies in an attempt to send a single buffer chunk to the underlying network write syscall. If 0, then no body is copied and a split write buffer will always be used. The default provided value (i.e. the non-overriden version) is unspecified (e.g. can change among versions and platforms).

4.2. Detailed

4.2.1. headers

#include <boost/http/headers.hpp>

headers is a simple typedef for some unspecified multimap container.

The user can safely assume that headers::key\_type will be std::string and headers::mapped\_type will be std::string. std::string is used because fulfills the requirements perfectly and is very unlikely it will ever cause any controversy.

The user can also assume that this type fulfills the [Headers definition of message](https://boostgsoc14.github.io/boost.http/#message_concept).

|  |  |
| --- | --- |
| **NOTE** | Previously, headers was guaranteed to be a typedef for boost::container::flat\_multimap. |

4.2.2. request

#include <boost/http/request.hpp>

request is a simple typedef for [basic\_request](https://boostgsoc14.github.io/boost.http/" \l "basic_request). It’s defined as follows:

typedef basic\_request<std::string, headers, std::vector<std::uint8\_t>> request;

std::vector<std::uint8\_t> is used over std::string, because fits the purpose of the body (binary data payload container) better (no '\0' character terminator, well-defined behaviours of capacity, size and iterator invalidation, …​).

4.2.2.1. See also

* [headers](https://boostgsoc14.github.io/boost.http/#headers)

4.2.3. response

#include <boost/http/response.hpp>

response is a simple typedef for [basic\_response](https://boostgsoc14.github.io/boost.http/" \l "basic_response). It’s defined as follows:

typedef basic\_response<std::string, headers, std::vector<std::uint8\_t>> response;

std::vector<std::uint8\_t> is used over std::string, because fits the purpose of the body (binary data payload container) better (no '\0' character terminator, well-defined behaviours of capacity, size and iterator invalidation, …​).

4.2.3.1. See also

* [headers](https://boostgsoc14.github.io/boost.http/#headers)

4.2.4. socket

#include <boost/http/socket.hpp>

socket is a simple typedef for [basic\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_socket). It’s defined as follows:

typedef basic\_socket<boost::asio::ip::tcp::socket> socket;

4.2.5. buffered\_socket

#include <boost/http/buffered\_socket.hpp>

buffered\_socket is a simple typedef for [basic\_buffered\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_buffered_socket). It’s defined as follows:

typedef basic\_buffered\_socket<boost::asio::ip::tcp::socket> buffered\_socket;

4.2.6. request\_response\_wrapper

#include <boost/http/request\_response\_wrapper.hpp>

request\_response\_wrapper is an adapter which acts like a common type (akin to std::common\_type) for different Request and Response types. There is no type erasure and only [Message interface](https://boostgsoc14.github.io/boost.http/#message_concept) will be available.

The purpose of this class is **not** to be a polymorphic wrapper. Therefore, it’s not our concern to make sure compatible objects (i.e. objects with the same header\_type and so on) are of the **same** request\_response\_wrapper type. In other words, request\_response\_wrapper will not free you from the template work if you want to support different Request/Response types. This is **not** a design issue. However, there is a templated constructor which can accept **some** different types.

4.2.6.1. Template parameters

Request

A model of the [Message concept](https://boostgsoc14.github.io/boost.http/#message_concept).

Response

A model of the [Message concept](https://boostgsoc14.github.io/boost.http/#message_concept).

|  |  |
| --- | --- |
| **NOTE** | Response::headers\_type must be the same as Request::headers\_type. And the same applies to Response::body\_type. Also, if const is applied to Request or Response, it must be applied to both. |

4.2.6.2. Member types

headers\_type

If Request is const, then it is defined as const typename Request::headers\_type. Otherwise, it’s defined as typename Request::headers\_type.

body\_type

If Request is const, then it is defined as const typename Request::body\_type. Otherwise, it’s defined as typename Request::body\_type.

4.2.6.3. Member functions

request\_response\_wrapper(Request &request)

Constructs a request\_response\_wrapper from a Request.

request\_response\_wrapper(Response &response)

Constructs a request\_response\_wrapper from a Response.

template<class Request2, class Response2> request\_response\_wrapper(request\_response\_wrapper<Request2, Response2> &other)

Constructs a request\_response\_wrapper from another request\_response\_wrapper with compatible headers\_type and body\_type.

Message concept

headers\_type &headers()

Returns the wrapped headers object.

const headers\_type &headers() const

Returns the wrapped headers object.

body\_type &body()

Returns the wrapped body object.

const body\_type &body() const

Returns the wrapped body object.

headers\_type &trailers()

Returns the wrapped trailers object.

const headers\_type &trailers() const

Returns the wrapped trailers object.

4.2.6.4. See also

* [poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#poly_socket_base)

4.2.7. basic\_poly\_socket\_base

#include <boost/http/poly\_socket\_base.hpp>

basic\_poly\_socket\_base is the base class for all classes in the hierarchy defined for runtime-based polymorphic HTTP producers. It is an abstract class that only contains functionality useful for simultaneously both channel ends (client and server).

References for objects of this class are expected to fulfill the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept).

This class has no state to ease multiple inheritance and it is virtual-inherited by [basic\_poly\_server\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_server_socket) and [basic\_poly\_client\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_client_socket).

The design for the hierarchy started with this class was a little inspired by *C++'s iostream* and [N3525: Polymorphic Allocators](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3525.pdf).

*Figure 1. poly\_socket\_base hierarchy*

4.2.7.1. Template parameters

Message

The message type.

4.2.7.2. Member types

These are the types chosen *set in stone* to guarantee ABI stability across binaries (including possible plugins).

typedef boost::asio::executor executor\_type

The executor type.

typedef Message message\_type

The message type usable within this class’s operations.

typedef asio::experimental::poly\_handler<void(boost::system::error\_code)> handler\_type

The type for asynchronous operation completion handlers.

4.2.7.3. Member functions

Overwritable functions

These are the functions that subclasses need to implement in order to lose the abstract class property.

virtual executor\_type get\_executor() = 0

This function presents no differences (besides mandatory virtual) from the one with the same name found on the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept).

virtual bool is\_open() const = 0

This function presents no differences (besides mandatory virtual) from the one with the same name found on the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept).

virtual read\_state read\_state() const = 0

This function presents no differences (besides mandatory virtual) from the one with the same name found on the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept).

virtual write\_state write\_state() const = 0

This function presents no differences (besides mandatory virtual) from the one with the same name found on the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept).

virtual void async\_read\_some(message\_type &message, handler\_type handler) = 0

The only difference between this function (besides mandatory virtual) and the one with the same name found on the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept) is the lack of support for completion tokens and the use of a type erased handler instead.

virtual void async\_read\_trailers(message\_type &message, handler\_type handler) = 0

The only difference between this function (besides mandatory virtual) and the one with the same name found on the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept) is the lack of support for completion tokens and the use of a type erased handler instead.

virtual void async\_write(const message\_type &message, handler\_type handler) = 0

The only difference between this function (besides mandatory virtual) and the one with the same name found on the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept) is the lack of support for completion tokens and the use of a type erased handler instead.

virtual void async\_write\_trailers(const message\_type &message, handler\_type handler) = 0

The only difference between this function (besides mandatory virtual) and the one with the same name found on the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept) is the lack of support for completion tokens and the use of a type erased handler instead.

virtual void async\_write\_end\_of\_message(handler\_type handler) = 0

The only difference between this function (besides mandatory virtual) and the one with the same name found on the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept) is the lack of support for completion tokens and the use of a type erased handler instead.

virtual ~basic\_poly\_socket\_base() = 0

Destructor. Inline definition done with = default.

Wrappers to fulfill the ASIO extensible model

These functions rewrite usual function calls in terms of the ABI stable interface. They also enable the ASIO extensible model within this hierarchy.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_read\_some(message\_type &message, CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_read\_some.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_read\_trailers(message\_type &message, CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_read\_trailers.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_write(const message\_type &message, CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_write.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_write\_trailers(const message\_type &message, CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_write\_trailers.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_write\_end\_of\_message(CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_write\_end\_of\_message.

4.2.7.4. See also

* [poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#poly_socket_base)

4.2.8. basic\_poly\_server\_socket

#include <boost/http/poly\_server\_socket.hpp>

This class virtual-inherits [basic\_poly\_socket\_base<Message>](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_socket_base) and extends its interface with virtual functions useful for server sockets. This class itself is also an abstract class with no state (i.e. an *interface*).

References for objects of this class are expected to fulfill the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept).

This class is inherited by [basic\_poly\_socket<Request, Response, Message>](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_socket), [server\_socket\_adaptor](https://boostgsoc14.github.io/boost.http/" \l "server_socket_adaptor) and [socket\_adaptor](https://boostgsoc14.github.io/boost.http/" \l "socket_adaptor).

4.2.8.1. Template parameters

Request

The request type.

Response

The response type.

Message = request\_response\_wrapper<Request, Response>

The message type.

4.2.8.2. Member types

These are the types chosen *set in stone* to guarantee ABI stability across binaries (including possible plugins).

typedef Request request\_type

The request type.

typedef Response response\_type

The response type.

4.2.8.3. Member functions

Overwritable functions

These are the functions that subclasses need to implement in order to lose the abstract class property.

virtual bool write\_response\_native\_stream() const = 0

This function presents no differences (besides mandatory virtual) from the one with the same name found on the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept).

virtual void async\_read\_request(request\_type &request, handler\_type handler) = 0

The only difference between this function (besides mandatory virtual) and the one with the same name found on the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept) is the lack of support for completion tokens and the use of a type erased handler instead.

virtual void async\_write\_response(const response\_type &response, handler\_type handler) = 0

The only difference between this function (besides mandatory virtual) and the one with the same name found on the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept) is the lack of support for completion tokens and the use of a type erased handler instead.

virtual void async\_write\_response\_continue(handler\_type handler) = 0

The only difference between this function (besides mandatory virtual) and the one with the same name found on the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept) is the lack of support for completion tokens and the use of a type erased handler instead.

virtual void async\_write\_response\_metadata(const response\_type &response, handler\_type handler) = 0

The only difference between this function (besides mandatory virtual) and the one with the same name found on the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept) is the lack of support for completion tokens and the use of a type erased handler instead.

virtual ~basic\_poly\_server\_socket()

Destructor.

Wrappers to fulfill the ASIO extensible model

These functions rewrite usual function calls in terms of the ABI stable interface. They also enable the ASIO extensible model within this hierarchy.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_read\_request(request\_type &request, CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_read\_request.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_write\_response(const response\_type &response, CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_write\_response.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_write\_response\_continue(CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_write\_response\_continue.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_write\_response\_metadata(const response\_type &response, CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_write\_response\_metadata.

4.2.8.4. See also

* [poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#poly_server_socket)

4.2.9. basic\_poly\_client\_socket

#include <boost/http/poly\_client\_socket.hpp>

This class virtual-inherits [basic\_poly\_socket\_base<Message>](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_socket_base) and extends its interface with virtual functions useful for client sockets. This class itself is also an abstract class with no state (i.e. an *interface*).

References for objects of this class are expected to fulfill the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept).

This class is inherited by [basic\_poly\_socket<Request, Response, Message>](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_socket), [client\_socket\_adaptor](https://boostgsoc14.github.io/boost.http/" \l "client_socket_adaptor) and [socket\_adaptor](https://boostgsoc14.github.io/boost.http/" \l "socket_adaptor).

4.2.9.1. Template parameters

Request

The request type.

Response

The response type.

Message = request\_response\_wrapper<Request, Response>

The message type.

4.2.9.2. Member types

These are the types chosen *set in stone* to guarantee ABI stability across binaries (including possible plugins).

typedef Request request\_type

The request type.

typedef Response response\_type

The response type.

4.2.9.3. Member functions

Overwritable functions

These are the functions that subclasses need to implement in order to lose the abstract class property.

virtual void async\_write\_request(const request\_type &request, handler\_type handler) = 0

This function presents no differences (besides mandatory virtual) from the one with the same name found on the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept).

virtual void async\_write\_request\_metadata(const request\_type &request, handler\_type handler) = 0

This function presents no differences (besides mandatory virtual) from the one with the same name found on the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept).

virtual void async\_read\_response(response\_type &response, handler\_type handler) = 0

This function presents no differences (besides mandatory virtual) from the one with the same name found on the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept).

virtual ~basic\_poly\_client\_socket()

Destructor.

Wrappers to fulfill the ASIO extensible model

These functions rewrite usual function calls in terms of the ABI stable interface. They also enable the ASIO extensible model within this hierarchy.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_write\_request(const request\_type &request, CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_write\_request.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_write\_request\_metadata(const request\_type &request, CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_write\_request\_metadata.

template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_read\_response(response\_type &response, CompletionToken &&token)

Handle the token and dispatch the operation to the ABI stable async\_read\_response.

4.2.9.4. See also

* [poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#poly_client_socket)

4.2.10. basic\_poly\_socket

#include <boost/http/poly\_socket.hpp>

This class inherits [basic\_poly\_server\_socket<Request, Response, Message>](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_server_socket) and [basic\_poly\_client\_socket<Request, Response, Message>](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_client_socket). This class itself is also an abstract class with no state (i.e. an *interface*).

References for objects of this class are expected to fulfill the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept) and the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept).

This class is inherited by [socket\_adaptor](https://boostgsoc14.github.io/boost.http/" \l "socket_adaptor).

4.2.10.1. Template parameters

Request

The request type.

Response

The response type.

Message = request\_response\_wrapper<Request, Response>

The message type.

4.2.10.2. Member types

These are the types chosen *set in stone* to guarantee ABI stability across binaries (including possible plugins).

typedef Request request\_type

Request message type.

typedef Response response\_type

Response message type.

4.2.10.3. See also

* [poly\_socket](https://boostgsoc14.github.io/boost.http/#poly_socket)

4.2.11. poly\_socket\_base

#include <boost/http/poly\_socket\_base.hpp>

poly\_socket\_base is a simple typedef for [basic\_poly\_socket\_base](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_socket_base). It’s defined as follows:

typedef basic\_poly\_socket\_base<request\_response\_wrapper<request, response>>

poly\_socket\_base;

4.2.11.1. See also

* [basic\_poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#basic_poly_socket_base)
* [request](https://boostgsoc14.github.io/boost.http/#request)
* [response](https://boostgsoc14.github.io/boost.http/#response)
* [request\_response\_wrapper](https://boostgsoc14.github.io/boost.http/#request_response_wrapper)

4.2.12. poly\_server\_socket

#include <boost/http/poly\_server\_socket.hpp>

poly\_server\_socket is a simple typedef for [basic\_poly\_server\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_server_socket). It’s defined as follows:

typedef basic\_poly\_server\_socket<request, response>

poly\_server\_socket;

4.2.12.1. See also

* [basic\_poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_server_socket)
* [message](https://boostgsoc14.github.io/boost.http/#message)

4.2.13. poly\_client\_socket

#include <boost/http/poly\_client\_socket.hpp>

poly\_client\_socket is a simple typedef for [basic\_poly\_client\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_client_socket). It’s defined as follows:

typedef basic\_poly\_client\_socket<request, response>

poly\_client\_socket;

4.2.13.1. See also

* [basic\_poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_client_socket)
* [message](https://boostgsoc14.github.io/boost.http/#message)

4.2.14. poly\_socket

#include <boost/http/poly\_socket.hpp>

poly\_socket is a simple typedef for [basic\_poly\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_socket). It’s defined as follows:

typedef basic\_poly\_socket<request, response> poly\_socket;

4.2.14.1. See also

* [basic\_poly\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_socket)
* [message](https://boostgsoc14.github.io/boost.http/#message)

4.2.15. basic\_request

#include <boost/http/request.hpp>

This template can be used to easily define classes fulfilling the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept) and specializing the [is\_request\_message](https://boostgsoc14.github.io/boost.http/" \l "is_request_message) trait.

4.2.15.1. Template parameters

String

The type to fulfill the X::string\_type from the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

Headers

The type to fulfill the X::headers\_type from the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

Body

The type to fulfill the X::body\_type from the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

4.2.15.2. Member types

typedef String string\_type

The type to fulfill the X::string\_type from the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

typedef Headers headers\_type

The type to fulfill the X::headers\_type from the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

typedef Body body\_type

The type to fulfill the X::body\_type from the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

4.2.15.3. Member functions

string\_type &method()

Returns the internal method object.

const string\_type &method() const

Returns the internal method object.

string\_type &target()

Returns the internal request target object.

const string\_type &target() const

Returns the internal request target object.

headers\_type &headers()

Returns the internal headers object.

const headers\_type &headers() const

Returns the internal headers object.

body\_type &body()

Returns the internal body object.

const body\_type &body() const

Returns the internal body object.

headers\_type &trailers()

Returns the internal trailers object.

const headers\_type &trailers() const

Returns the internal trailers object.

4.2.15.4. See also

* [request](https://boostgsoc14.github.io/boost.http/#request)

4.2.16. basic\_response

#include <boost/http/response.hpp>

This template can be used to easily define classes fulfilling the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept) and specializing the [is\_response\_message](https://boostgsoc14.github.io/boost.http/" \l "is_response_message) trait.

4.2.16.1. Template parameters

String

The type to fulfill the X::string\_type from the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

Headers

The type to fulfill the X::headers\_type from the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

Body

The type to fulfill the X::body\_type from the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

4.2.16.2. Member types

typedef String string\_type

The type to fulfill the X::string\_type from the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

typedef Headers headers\_type

The type to fulfill the X::headers\_type from the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

typedef Body body\_type

The type to fulfill the X::body\_type from the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

4.2.16.3. Member functions

std::uint\_least16\_t &status\_code()

Returns the internal status code object.

const std::uint\_least16\_t &status\_code() const

Returns the internal status code object.

string\_type &reason\_phrase()

Returns the internal reason phrase object.

const string\_type &reason\_phrase() const

Returns the internal reason phrase object.

headers\_type &headers()

Returns the internal headers object.

const headers\_type &headers() const

Returns the internal headers object.

body\_type &body()

Returns the internal body object.

const body\_type &body() const

Returns the internal body object.

headers\_type &trailers()

Returns the internal trailers object.

const headers\_type &trailers() const

Returns the internal trailers object.

4.2.16.4. See also

* [response](https://boostgsoc14.github.io/boost.http/#response)

4.2.17. basic\_socket

#include <boost/http/socket.hpp>

This template can be used to easily define classes fulfilling the strict [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept) and the strict [ClientSocket concept](https://boostgsoc14.github.io/boost.http/#client_socket_concept) ([is\_server\_socket](https://boostgsoc14.github.io/boost.http/#is_server_socket) and [is\_client\_socket](https://boostgsoc14.github.io/boost.http/#is_client_socket) traits are specialized). These classes exposes the HTTP/1.1 wire format (i.e. a builtin/standalone HTTP server) into an easy-to-use API.

The underlying I/O object is expected to have the following properties:

* It is stream-oriented (i.e. no message boundaries; read or write operations may transfer fewer bytes than requested…​).
* It fulfills the ASIO’s AsyncReadStream requirement.
* It fulfills the ASIO’s AsyncWriteStream requirement.
* It is backed by a reliable transport or session-layer “connection” with in-order delivery of octets (i.e. any 8-bit sequence of data).

|  |  |  |
| --- | --- | --- |
| **NOTE** | This class doesn’t restrict the message, method, path and reason phrase types. Therefore, the following members are **not** defined:   * message\_type * request\_type * response\_type | |
| **WARNING** | | The API from this class is implemented in terms of composed operations. As such, you MUST **NOT** initiate any async read operation while there is another read operation in progress and you MUST **NOT** initiate any async write operation while there is another write operation in progress. If you cannot guarantee the ordering of the operations, you should use some queueing socket (e.g. [AxioMQ’s basic\_queue\_socket](http://sourceforge.net/p/axiomq/code/ci/master/tree/include/axiomq/basic_queue_socket.hpp)). |

|  |  |
| --- | --- |
| **TIP** | You cannot detect the lack of network inactivity properly under this layer. If you need to implement timeouts, you should do so under the lower layer. |

4.2.17.1. Template parameters

Socket

The underlying communication channel type. It MUST fulfill the requirements for ASIO’s AsyncReadStream and ASIO’s AsyncWriteStream.

Settings

Traits to define some settings. It defaults to *unspecified*.

As a reference, the default traits class has the following form:

struct

{

typedef reader::request req\_parser;

typedef reader::response res\_parser;

static constexpr std::size\_t body\_copy\_threshold

= BOOST\_HTTP\_SOCKET\_DEFAULT\_BODY\_COPY\_THRESHOLD;

};

4.2.17.2. Member types

typedef Socket next\_layer\_type

The type of the underlying communication channel.

typedef typename next\_layer\_type::executor\_type executor\_type

The type of the executor associated with the object.

4.2.17.3. Member functions

basic\_socket(boost::asio::io\_context &io\_context, boost::asio::mutable\_buffer inbuffer)

Constructor. *io\_context* is passed to the constructor from the underlying stream.

*Exceptions:*

* std::invalid\_argument: If buffer size is zero.

template<class…​ Args> basic\_socket(boost::asio::mutable\_buffer inbuffer, Args&&…​ args)

Constructor. *args* are forwarded to the constructor from the underlying stream.

*Exceptions:*

* std::invalid\_argument: If buffer size is zero.

next\_layer\_type &next\_layer()

Returns a reference to the underlying stream.

const next\_layer\_type &next\_layer() const

Returns a reference to the underlying stream.

void open()

Change socket state to open.

|  |  |  |
| --- | --- | --- |
| **NOTE** | See is\_open() | |
| **WARNING** | | You MUST cancel current ongoing operations and wait for their completion handlers to be called before call this function. Otherwise, undefined behaviour is invoked. |

boost::asio::const\_buffer upgrade\_head() const

Return the buffer representing the first few bytes of the upgraded stream (may be empty).

void lock\_client\_to\_http10()

Lock HTTP to HTTP/1.0 version if socket is used as a client socket.

template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken,void(system::error\_code, std::size\_t)) async\_read\_chunkext(Message &message, typename Message::headers\_type &chunkext, CompletionToken &&token)

Initiate an asynchronous operation to read a part of the message body. Handler is called when at least one byte is read (called with no error set and value 0), when chunk extensions are found (called with no error set and some value different than 0), when the end of message is reached (called with no error set and value 0) or when some error occurs.

message.body(), message.trailers() and chunkext are left in an unspecified state while the operation is in progress. chunkext.clear() will be called by this function before the read operation is scheduled.

When valid chunk extensions are found, they are stored in chunkext and handler is called before the chunk bytes are stored in message.body(). The chunk size is passed to the handler and chunkext data refers to the **next** bytes that will be pushed into message.body() (i.e. you need to initiate another read to get the chunk data referred by chunkext).

This design is simple (no need for extra error code, simple to explain, simple to program for, …​) and does expose relevant information to the user (e.g. you can extract message boundaries) without compromising security (read is still limited by buffer size and you will **not** be exposed to — for instance — infinite appends into message.body()).

template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_write\_chunkext(const Message &message, const typename Message::headers\_type &chunkext, CompletionToken &&token)

Initiate an asynchronous operation to write a chunk of the HTTP body data payload (chunked message). Handler is called with an appropriate argument when the operation completes.

message.body() and chunkext MUST **NOT** be modified while the operation is in progress.

Socket concept

See the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept).

* executor\_type get\_executor()
* bool is\_open() const
* read\_state read\_state() const
* write\_state write\_state() const
* template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_read\_some(Message &message, CompletionToken &&token)
* template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_read\_trailers(Message &message, CompletionToken &&token)
* template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write(const Message &message, CompletionToken &&token)
* template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_trailers(const Message &message, CompletionToken &&token)
* template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_end\_of\_message(CompletionToken &&token)

ServerSocket concept

See the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept).

* bool write\_response\_native\_stream() const
* template<class Request, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_read\_request(Request &request, CompletionToken &&token)
* template<class Response, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_response(const Response &response, CompletionToken &&token)
* template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_response\_continue(CompletionToken &&token)
* template<class Response, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_response\_metadata(const Response &response, CompletionToken &&token)

ClientSocket concept

See the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept).

* template<class Request, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_request(const Request &request, CompletionToken &&token)
* template<class Request, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_request\_metadata(const Request &request, CompletionToken &&token)
* template<class Response, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_read\_response(Response &response, CompletionToken &&token)

4.2.18. basic\_buffered\_socket

#include <boost/http/buffered\_socket.hpp>

This template can be used to easily define classes fulfilling the strict [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept) and [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept). These classes exposes the HTTP/1.1 wire format (i.e. a builtin/standalone HTTP server) into an easy-to-use API.

The underlying I/O object is expected to have the following properties:

* It is stream-oriented (i.e. no message boundaries; read or write operations may transfer fewer bytes than requested…​).
* It fulfills the ASIO’s AsyncReadStream requirement.
* It fulfills the ASIO’s AsyncWriteStream requirement.
* It is backed by a reliable transport or session-layer “connection” with in-order delivery of octets (i.e. any 8-bit sequence of data).

|  |  |  |
| --- | --- | --- |
| **NOTE** | This class doesn’t restrict the message, method, path and reason phrase types. Therefore, the following members are **not** defined:   * message\_type * request\_type * response\_type | |
| **WARNING** | | The API from this class is implemented in terms of composed operations. As such, you MUST **NOT** initiate any async read operation while there is another read operation in progress and you MUST **NOT** initiate any async write operation while there is another write operation in progress. If you cannot guarantee the ordering of the operations, you should use some queueing socket (e.g. [AxioMQ’s basic\_queue\_socket](http://sourceforge.net/p/axiomq/code/ci/master/tree/include/axiomq/basic_queue_socket.hpp)). |

|  |  |
| --- | --- |
| **TIP** | You cannot detect the lack of network inactivity properly under this layer. If you need to implement timeouts, you should do so under the lower layer. |

4.2.18.1. Template parameters

Socket

The underlying communication channel type. It MUST fulfill the requirements for ASIO’s AsyncReadStream and ASIO’s AsyncWriteStream.

Settings

Traits passed to the underlying Socket. It defaults to *unspecified*.

N

The internal buffer size. It defaults to BOOST\_HTTP\_SOCKET\_DEFAULT\_BUFFER\_SIZE

4.2.18.2. Member types

typedef Socket next\_layer\_type

The type of the underlying communication channel.

typedef typename next\_layer\_type::executor\_type executor\_type

The type of the executor associated with the object.

4.2.18.3. Member functions

basic\_buffered\_socket(boost::asio::io\_context &io\_context)

Constructor. *io\_context* is passed to the constructor from the underlying stream.

template<class…​ Args> basic\_buffered\_socket(Args&&…​ args)

Constructor. *args* are forwarded to the constructor from the underlying stream.

next\_layer\_type &next\_layer()

Returns a reference to the underlying stream.

const next\_layer\_type &next\_layer() const

Returns a reference to the underlying stream.

boost::asio::const\_buffer upgrade\_head() const

Return the buffer representing the first few bytes of the upgraded stream (may be empty).

void lock\_client\_to\_http10()

Lock HTTP to HTTP/1.0 version if socket is used as a client socket.

template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken,void(system::error\_code, std::size\_t)) async\_read\_chunkext(Message &message, typename Message::headers\_type &chunkext, CompletionToken &&token)

See documentation in [basic\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_socket).

template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(system::error\_code)) async\_write\_chunkext(const Message &message, const typename Message::headers\_type &chunkext, CompletionToken &&token)

See documentation in [basic\_socket](https://boostgsoc14.github.io/boost.http/" \l "basic_socket).

Socket concept

See the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept).

* executor\_type get\_executor()
* bool is\_open() const
* read\_state read\_state() const
* write\_state write\_state() const
* template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_read\_some(Message &message, CompletionToken &&token)
* template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_read\_trailers(Message &message, CompletionToken &&token)
* template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write(const Message &message, CompletionToken &&token)
* template<class Message, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_trailers(const Message &message, CompletionToken &&token)
* template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_end\_of\_message(CompletionToken &&token)

ServerSocket concept

See the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept).

* bool write\_response\_native\_stream() const
* template<class Request, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_read\_request(Request &request, CompletionToken &&token)
* template<class Response, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_response(const Response &response, CompletionToken &&token)
* template<class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_response\_continue(CompletionToken &&token)
* template<class Response, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_response\_metadata(const Response &response, CompletionToken &&token)

ClientSocket concept

See the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept).

* template<class Request, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_request(const Request &request, CompletionToken &&token)
* template<class Request, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_write\_request\_metadata(const Request &request, CompletionToken &&token)
* template<class Response, class CompletionToken> BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code)) async\_read\_response(Response &response, CompletionToken &&token)

4.2.19. server\_socket\_adaptor

#include <boost/http/server\_socket\_adaptor.hpp>

This class adapts a class fulfilling the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept) to implement the [basic\_poly\_server\_socket<Request, Response, Message>](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_server_socket) interface.

This design was chosen taking a number of shortcomings in consideration.

If the user can control object construction, he might benefit by less levels of indirection by constructing both (the HTTP socket and its runtime-based polymorphic adaptor) at once and at a single memory space.

The scenario where the user don’t control the object construction was also taken in consideration. In these cases, it’s possible to use the std::reference\_wrapper type found in the <functional> header. There is a server\_socket\_adaptor specialization that will do the job for std::reference\_wrapper.

Also, if the user needs to query for the specific type at runtime, the user can do so with a single call to dynamic\_cast to the specific polymorphic wrapper (rather than calling a second function to query for the wrapped object).

The design is simple to use, to learn and to read. These values were chosen to avoid misuse by the user’s part.

Although very different, the name and inspiration were borrowed from [N3525: Polymorphic Allocators](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3525.pdf).

4.2.19.1. Template parameters

Socket

The type to be wrapped.

Request = request

The request message type. Default argument is [request](https://boostgsoc14.github.io/boost.http/#request).

Response = response

The response message type. Default argument is [response](https://boostgsoc14.github.io/boost.http/#response).

Message = request\_response\_wrapper<Request, Response>

The message type. Default argument is [request\_response\_wrapper<Request, Response>](https://boostgsoc14.github.io/boost.http/" \l "request_response_wrapper).

4.2.19.2. Specializations

* server\_socket\_adaptor<std::reference\_wrapper<Socket>>

4.2.19.3. Member types

typedef Socket next\_layer\_type

The type of the wrapped socket.

4.2.19.4. Member functions

template<class…​ Args> server\_socket\_adaptor(Args&&…​ args)

Constructor. args are forwarded to the underlying socket constructor.

|  |  |
| --- | --- |
| **NOTE** | Not available under the server\_socket\_adaptor<std::reference\_wrapper<Socket>> specialization. |

server\_socket\_adaptor(Socket &socket)

Constructor. socket is passed to the std::reference\_wrapper constructor.

|  |  |
| --- | --- |
| **NOTE** | Only available under the server\_socket\_adaptor<std::reference\_wrapper<Socket>> specialization. |

next\_layer\_type &next\_layer()

Socket isn’t exposed directly to avoid confusion over the duplication of interfaces.

The name socket is not used because both (the wrapped object and this object itself) are sockets and it would be confusing.

const next\_layer\_type &next\_layer() const

Socket isn’t exposed directly to avoid confusion over the duplication of interfaces.

The name socket is not used because both (the wrapped object and this object itself) are sockets and it would be confusing.

4.2.20. client\_socket\_adaptor

#include <boost/http/client\_socket\_adaptor.hpp>

This class adapts a class fulfilling the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept) to implement the [basic\_poly\_client\_socket<Request, Response, Message>](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_client_socket) interface.

This design was chosen taking a number of shortcomings in consideration.

If the user can control object construction, he might benefit by less levels of indirection by constructing both (the HTTP socket and its runtime-based polymorphic adaptor) at once and at a single memory space.

The scenario where the user doesn’t control the object construction was also taken in consideration. In these cases, it’s possible to use the std::reference\_wrapper type found in the <functional> header. There is a client\_socket\_adaptor specialization that will do the job for std::reference\_wrapper.

Also, if the user needs to query for the specific type at runtime, the user can do so with a single call to dynamic\_cast to the specific polymorphic wrapper (rather than calling a second function to query for the wrapped object).

The design is simple to use, to learn and to read. These values were chosen to avoid misuse by the user’s part.

Although very different, the name and inspiration were borrowed from [N3525: Polymorphic Allocators](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3525.pdf).

4.2.20.1. Template parameters

Socket

The type to be wrapped.

Request = request

The request message type. Default argument is [request](https://boostgsoc14.github.io/boost.http/#request).

Response = response

The response message type. Default argument is [response](https://boostgsoc14.github.io/boost.http/#response).

Message = request\_response\_wrapper<Request, Response>

The message type. Default argument is [request\_response\_wrapper<Request, Response>](https://boostgsoc14.github.io/boost.http/" \l "request_response_wrapper).

4.2.20.2. Specializations

* client\_socket\_adaptor<std::reference\_wrapper<Socket>>

4.2.20.3. Member types

typedef Socket next\_layer\_type

The type of the wrapped socket.

4.2.20.4. Member functions

template<class…​ Args> client\_socket\_adaptor(Args&&…​ args)

Constructor. args are forwarded to the underlying socket constructor.

|  |  |
| --- | --- |
| **NOTE** | Not available under the client\_socket\_adaptor<std::reference\_wrapper<Socket>> specialization. |

client\_socket\_adaptor(Socket &socket)

Constructor. socket is passed to the std::reference\_wrapper constructor.

|  |  |
| --- | --- |
| **NOTE** | Only available under the client\_socket\_adaptor<std::reference\_wrapper<Socket>> specialization. |

next\_layer\_type &next\_layer()

Socket isn’t exposed directly to avoid confusion over the duplication of interfaces.

The name socket is not used because both (the wrapped object and this object itself) are sockets and it would be confusing.

const next\_layer\_type &next\_layer() const

Socket isn’t exposed directly to avoid confusion over the duplication of interfaces.

The name socket is not used because both (the wrapped object and this object itself) are sockets and it would be confusing.

4.2.21. socket\_adaptor

#include <boost/http/socket\_adaptor.hpp>

This class adapts a class fulfilling the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept) and the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept) to implement the [basic\_poly\_socket<Request, Response, Message>](https://boostgsoc14.github.io/boost.http/" \l "basic_poly_socket) interface.

This design was chosen taking a number of shortcomings in consideration.

If the user can control object construction, he might benefit by less levels of indirection by constructing both (the HTTP socket and its runtime-based polymorphic adaptor) at once and at a single memory space.

The scenario where the user doesn’t control the object construction was also taken into consideration. In these cases, it’s possible to use the std::reference\_wrapper type found in the <functional> header. There is a socket\_adaptor specialization that will do the job for std::reference\_wrapper.

Also, if the user needs to query for the specific type at runtime, the user can do so with a single call to dynamic\_cast to the specific polymorphic wrapper (rather than calling a second function to query for the wrapped object).

The design is simple to use, to learn and to read. These values were chosen to avoid misuse by the user’s part.

Although very different, the name and inspiration were borrowed from [N3525: Polymorphic Allocators](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3525.pdf).

4.2.21.1. Template parameters

Socket

The type to be wrapped.

Request = request

The request message type. Default argument is [request](https://boostgsoc14.github.io/boost.http/#request).

Response = response

The response message type. Default argument is [response](https://boostgsoc14.github.io/boost.http/#response).

Message = request\_response\_wrapper<Request, Response>

The message type. Default argument is [request\_response\_wrapper<Request, Response>](https://boostgsoc14.github.io/boost.http/" \l "request_response_wrapper).

4.2.21.2. Specializations

* socket\_adaptor<std::reference\_wrapper<Socket>>

4.2.21.3. Member types

typedef Socket next\_layer\_type

The type of the wrapped socket.

4.2.21.4. Member functions

template<class…​ Args> socket\_adaptor(Args&&…​ args)

Constructor. args are forwarded to the underlying socket constructor.

|  |  |
| --- | --- |
| **NOTE** | Not available under the socket\_adaptor<std::reference\_wrapper<Socket>> specialization. |

socket\_adaptor(Socket &socket)

Constructor. socket is passed to the std::reference\_wrapper constructor.

|  |  |
| --- | --- |
| **NOTE** | Only available under the socket\_adaptor<std::reference\_wrapper<Socket>> specialization. |

next\_layer\_type &next\_layer()

Socket isn’t exposed directly to avoid confusion over the duplication of interfaces.

The name socket is not used because both (the wrapped object and this object itself) are sockets and it would be confusing.

const next\_layer\_type &next\_layer() const

Socket isn’t exposed directly to avoid confusion over the duplication of interfaces.

The name socket is not used because both (the wrapped object and this object itself) are sockets and it would be confusing.

4.2.22. header\_to\_ptime

#include <boost/http/algorithm/header.hpp>

template<class StringView>

boost::posix\_time::ptime header\_to\_ptime(const StringView &value)

Converts an *HTTP-date* [[15](https://boostgsoc14.github.io/boost.http/#_footnotedef_15)] field value into boost::posix\_time::ptime.

|  |  |
| --- | --- |
| **NOTE** | Values containing extra whitespace at the beginning or at the end of *value* will be rejected and no conversion will be done. This behaviour is intentional. |

4.2.22.1. Template parameters

StringView

It MUST fulfill the requirements of the StringView concept (i.e. boost::basic\_string\_view).

4.2.22.2. Paremeters

const StringView &value

An *HTTP-date*.

4.2.22.3. Return value

The converted value if *value* is a valid *HTTP-date* or boost::posix\_time::ptime(date\_time::not\_a\_date\_time) otherwise.

|  |  |
| --- | --- |
| **TIP** | You can use ptime’s `is\_not\_a\_date\_time() member-function to check if the conversion failed. |

4.2.22.4. See also

* [to\_http\_date](https://boostgsoc14.github.io/boost.http/#to_http_date)

4.2.23. to\_http\_date

#include <boost/http/algorithm/header.hpp>

template<class String>

String to\_http\_date(const boost::posix\_time::ptime &datetime)

Converts a boost::posix\_time::ptime into the preferred string representation according to section 7.1.1.1 of RFC 7231 (i.e. fixed length/zone/capitalization subset of the format defined in section 3.3 of RFC 5322).

4.2.23.1. Template parameters

String

It MUST fulfill the requirements of the String concept (i.e. std::basic\_string).

4.2.23.2. Parameters

const boost::posix\_time::ptime &datetime

The timepoint to be converted. It MUST be in UTC timezone.

4.2.23.3. Return value

The string representation in the preferred format (a.k.a. *IMF-fixdate*).

4.2.23.4. Exceptions

* std::out\_of\_range: If invalid *datetime* is given.

4.2.23.5. See also

* [header\_to\_ptime](https://boostgsoc14.github.io/boost.http/#header_to_ptime)

4.2.24. header\_value\_all\_of

#include <boost/http/algorithm/header.hpp>

template<class StringView, class Predicate>

bool header\_value\_all\_of(const StringView &header\_value, const Predicate &p)

Checks if unary predicate *p* returns true for all elements from the comma-separated list defined by the *header\_value* HTTP field value.

|  |  |
| --- | --- |
| **NOTE** | This algorithm is liberal in what it accepts and it will skip invalid elements. An invalid element is a sequence, possibly empty, containing no other character than optional white space (i.e. '\x20' or '\t'). |

4.2.24.1. Template parameters

StringView

It MUST fulfill the requirements of the StringView concept (i.e. boost::basic\_string\_view).

Predicate

A type whose instances are callable and have the following signature:

bool(StringView)

4.2.24.2. Parameters

const StringView &header\_value

The HTTP field value.

const Predicate &p

The functor predicate that will be called for the elements found on the comma-separated list.

Optional white space (only at the beginning and at the end) is trimmed before applying the element to *p*.

4.2.24.3. Return value

true if *p* doesn’t returns false for any element from the list and false otherwise. This also means that you’ll get the return value true for empty lists.

4.2.25. header\_value\_any\_of

#include <boost/http/algorithm/header.hpp>

template<class StringView, class Predicate>

bool header\_value\_any\_of(const StringView &header\_value, const Predicate &p)

Checks if unary predicate *p* returns true for at least one element from the comma-separated list defined by the *header\_value* HTTP field value.

|  |  |
| --- | --- |
| **NOTE** | This algorithm is liberal in what it accepts and it will skip invalid elements. An invalid element is a sequence, possibly empty, containing no other character than optional white space (i.e. '\x20' or '\t'). |

4.2.25.1. Template parameters

StringView

It MUST fulfill the requirements of the StringView concept (i.e. boost::basic\_string\_view).

Predicate

A type whose instances are callable and have the following signature:

bool(StringView)

4.2.25.2. Parameters

const StringView &header\_value

The HTTP field value.

const Predicate &p

The functor predicate that will be called for the elements found on the comma-separated list.

Optional white space (only at the beginning and at the end) is trimmed before applying the element to *p*.

4.2.25.3. Return value

true if the *p* returns true for at least one element from the list and false otherwise. This also means that you’ll get the return value false for empty lists.

4.2.26. header\_value\_none\_of

#include <boost/http/algorithm/header.hpp>

template<class StringView, class Predicate>

bool header\_value\_none\_of(const StringView &header\_value, const Predicate &p)

Checks if unary predicate *p* returns true for no elements from the comma-separated list defined by the *header\_value* HTTP field value.

|  |  |
| --- | --- |
| **NOTE** | This algorithm is liberal in what it accepts and it will skip invalid elements. An invalid element is a sequence, possibly empty, containing no other character than optional white space (i.e. '\x20' or '\t'). |

4.2.26.1. Template parameters

StringView

It MUST fulfill the requirements of the StringView concept (i.e. boost::basic\_string\_view).

Predicate

A type whose instances are callable and have the following signature:

bool(StringView)

4.2.26.2. Parameters

const StringView &header\_value

The HTTP field value.

const Predicate &p

The functor predicate that will be called for the elements found on the comma-separated list.

Optional white space (only at the beginning and at the end) is trimmed before applying the element to *p*.

4.2.26.3. Return value

true if *p* doesn’t returns true for any element from the list and false otherwise. This also means that you’ll get the return value true for empty lists.

4.2.27. header\_value\_for\_each

#include <boost/http/algorithm/header.hpp>

template<class StringView, class F>

F header\_value\_for\_each(const StringView &header\_value, F f)

Apply *f* for each element from the comma-separated list defined by the *header\_value* HTTP field value.

|  |  |
| --- | --- |
| **NOTE** | This algorithm is liberal in what it accepts and it will skip invalid elements. An invalid element is a sequence, possibly empty, containing no other character than optional white space (i.e. '\x20' or '\t'). |

4.2.27.1. Template parameters

StringView

It MUST fulfill the requirements of the StringView concept (i.e. boost::basic\_string\_view).

F

A type whose instances are callable and have the following signature:

void(StringView)

4.2.27.2. Parameters

const StringView &header\_value

The HTTP field value.

F f

The functor that will be called for the elements found on the comma-separated list.

Optional white space (only at the beginning and at the end) is trimmed before applying the element to *f*.

4.2.27.3. Return value

std::move(f)

4.2.28. etag\_match\_strong

#include <boost/http/algorithm/header.hpp>

template<class StringView>

bool etag\_match\_strong(const StringView &a, const StringView &b)

Check if *a* and *b* match.

|  |  |
| --- | --- |
| **NOTE** | This function doesn’t pedantically check if both arguments are actual entity tags and you should validate at least one of the arguments yourself.  Builtin validation is only done enough to protect against attacks. |

4.2.28.1. Template parameters

StringView

It MUST fulfill the requirements of the StringView concept (i.e. boost::basic\_string\_view).

4.2.28.2. Parameters

const StringView &a

The entity tag to compare against *b*.

const StringView &b

The entity tag to compare against *a*.

4.2.28.3. Return value

true if the entity tags match, using the strong comparison [[16](https://boostgsoc14.github.io/boost.http/#_footnotedef_16)] or false otherwise.

4.2.29. etag\_match\_weak

#include <boost/http/algorithm/header.hpp>

template<class StringView>

bool etag\_match\_weak(const StringView &a, const StringView &b)

Check if *a* and *b* match.

|  |  |
| --- | --- |
| **NOTE** | This function doesn’t pedantically check if both arguments are actual entity tags and you should validate at least one of the arguments yourself.  Builtin validation is only done enough to protect against attacks. |

4.2.29.1. Template parameters

StringView

It MUST fulfill the requirements of the StringView concept (i.e. boost::basic\_string\_view).

4.2.29.2. Parameters

const StringView &a

The entity tag to compare against *b*.

const StringView &b

The entity tag to compare against *a*.

4.2.29.3. Return value

true if the entity tags match, using the weak comparison [[17](https://boostgsoc14.github.io/boost.http/#_footnotedef_17)] or false otherwise.

4.2.30. request\_continue\_required

#include <boost/http/algorithm/query.hpp>

template<class Request>

bool request\_continue\_required(const Request &request)

Check if the request represented by *request* requires a “100 (Continue) response” [[18](https://boostgsoc14.github.io/boost.http/#_footnotedef_18)].

If you can properly process and reply the message without its body, you’re free to go. Otherwise, you should send a “100 (Continue) response” to ask for the message body from the HTTP client.

This feature was designed to decrease network traffic, by allowing servers to sooner reject messages that would be discarded anyway.

The name *required* is used instead *supported*, because an action from the server is required.

4.2.30.1. Template parameters

Request

A type fulfilling the requirements for the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

4.2.30.2. Parameters

const Request &request

The read message.

4.2.30.3. Return value

Whether the request represented by *request* requires a “100 (Continue) response”.

4.2.31. request\_upgrade\_desired

#include <boost/http/algorithm/query.hpp>

template<class Request,

class StringView = boost::basic\_string\_view<

typename Request::headers\_type::mapped\_type::value\_type>>

bool request\_upgrade\_desired(const Request &request)

Check if the client desires to initiate a protocol upgrade.

The desired protocols are present in the "upgrade" header as a comma-separated list.

|  |  |
| --- | --- |
| **WARNING** | You MUST **NOT** upgrade to a protocol listed in the "upgrade" header if this function returns false. |

The upgrade desire can always be safely ignored.

The user MUST wait till the whole request is received before proceeding to the protocol upgrade.

4.2.31.1. Template parameters

Request

A type fulfilling the requirements for the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

StringView

A type fulfilling the requirements for the StringView concept (i.e. boost::basic\_string\_view).

4.2.31.2. Parameters

const Request &request

The read message.

4.2.31.3. Return value

Whether the client desires to initiate a protocol upgrade.

4.2.31.4. See also

* [header\_value\_any\_of](https://boostgsoc14.github.io/boost.http/#header_value_any_of)
* [header\_value\_for\_each](https://boostgsoc14.github.io/boost.http/#header_value_for_each)

4.2.32. async\_response\_transmit\_file

#include <boost/http/file\_server.hpp>

This function has two overloads.

template<class ServerSocket, class Request, class Response,

class CompletionToken>

BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code))

async\_response\_transmit\_file(ServerSocket &socket, const Request &imessage,

Response &omessage,

const boost::filesystem::path &file,

CompletionToken &&token); // (1)

template<class ServerSocket, class Request, class Response,

class CompletionToken>

BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code))

async\_response\_transmit\_file(ServerSocket &socket, const Request &imessage,

Response &omessage,

const boost::filesystem::path &file,

bool is\_head\_request,

CompletionToken &&token); // (2)

This function will handle a big part of the file serving job for you, such as:

* It’ll interpret and process the request headers. However, it doesn’t process the request method nor the the input path. This means you still have to guarantee the method is applicable and you’re also expected to resolve the input path to a valid local file path.
* It’ll fill all the applicable response headers appropriate to the request.
* It’ll interpret the file attributes to process conditional requests and partial download. However, MIME detection (the "content-type" optional header) is still left for the user to handle.

Summarizing your responsibilities:

* Ensure it is a "GET" method or a method with similar semantics before calling this function. *Overload 2* also accepts the "HEAD" method.
* Resolve the input URL to the appropriate file.
* **Optional**: MIME detection (the "content-type" header).
* **Optional**: ETag detection (see [below](https://boostgsoc14.github.io/boost.http/#async_response_transmit_file_etag)).

A method with semantics similar to the "GET" method is any method fulfilling the following conditions:

* The body data payload for the response message is compatible.
* The following headers for the request message have the same meaning:
  + "if-modified-since"
  + "if-range"
  + "if-unmodified-since"
  + "range"
* The following headers for the response message have the same meaning or don’t affect the message processing:
  + "accept-ranges"
  + "content-range"
  + "content-type"
  + "date"
  + "last-modified"
* The same set of status code for the response are applicable.
  + "200 OK"
  + "206 Partial Content"
  + "304 Not Modified"
  + "412 Precondition Failed"
  + "416 Range Not Satisfiable"

|  |  |
| --- | --- |
| **NOTE** | This function will call the handler with file\_server\_errc::io\_error if any operation on the **file stream** fails or throws an exception. |
| **NOTE** | The response headers filled by this function MUST NOT be sent through trailers. Therefore, this function will not do any operation and it will call the handler with an error\_code (file\_server\_errc::write\_state\_not\_supported) set if you pass a socket for which the headers were already sent. |

|  |  |  |
| --- | --- | --- |
| **CAUTION** | | async\_response\_transmit\_file will make use of the streaming interface only if available.  If you want to avoid wasting memory under HTTP/1.0 and other non-streaming capable channels, starting points for two solutions are:   * Reject channels without a streaming interface (e.g. HTTP/1.0). RFC 2068 (a.k.a. HTTP/1.1) has been available since January 1997. * Write a custom message type that adapts the filestream to the message concept and specialize the algorithm for such message. |
| **NOTE** | omessage.body() will be used as output buffer. If omessage.body().capacity() == 0, an unspecified buffer size will be used and it is very likely it’ll be highly inefficient. | |

4.2.32.1. ETags

An ETag is a string that identify a representation of a resource. The ETag can be used to perform conditional requests more robust than the ones done with dates (limited by HTTP to seconds-based precision). In the conditional request context, an etag is a validator.

If you want to make use of the ETag implementation, just set the "etag" header in the omessage object to the appropriate value, as described below (also described in more details in RFC7232).

The first decision you must do if you decide to provide an etag is if you’re going to provide a strong validator or a weak validator.

Strong validators

A strong validator changes whenever a change occurs to the representation data that would be observable in the payload body. A strong validator is unique across all versions of all representations associated with a particular resource over time.

A strong etag has the form (specified in Augmented Backus-Naur Form (ABNF) notation of [RFC5234]):

strong-etag = DQUOTE \*etagc DQUOTE

etagc = %x21 / %x23-7E / obs-text

; VCHAR except double quotes, plus obs-text

obs-text = %x80-FF

|  |  |
| --- | --- |
| **CAUTION** | Some recipients might perform backslash unescaping. Therefore, it is a good practice to avoid backslash characters. |

Some examples:

* "xyasdzzy"
* "xyz9czy"
* ""

|  |  |
| --- | --- |
| **NOTE** | A strong validator might change for reasons other than a change to the representation data. |
| **NOTE** | There is no implication of uniqueness across representations of different resources. |

Weak validators

A weak validator might not change for every change to the representation data.

A weak etag has the form (specified in Augmented Backus-Naur Form (ABNF) notation of [RFC5234]):

weak-etag = weak opaque-tag

weak = %x57.2F ; "W/", case-sensitive

opaque-tag = DQUOTE \*etagc DQUOTE

etagc = %x21 / %x23-7E / obs-text

; VCHAR except double quotes, plus obs-text

obs-text = %x80-FF

|  |  |
| --- | --- |
| **CAUTION** | Some recipients might perform backslash unescaping. Therefore, it is a good practice to avoid backslash characters. |

Some examples:

* W/"xyasdzzy"
* W/"xyz9czy"
* W/""

4.2.32.2. Template parameters

ServerSocket

Must fulfill the requirements for the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept).

Request

Must fulfill the requirements for the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

Response

Must fulfill the requirements for the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

|  |  |
| --- | --- |
| **CAUTION** | Response::body\_type MUST fulfill the following extra requirements:   * Its elements MUST be stored contiguously (e.g. std::vector). * It MUST support C++11 std::vector capicity and data semantics (*vector.capacity* and *vector.data*, respectively).   These extra requirements are posed because file APIs are defined in terms of buffer [[19](https://boostgsoc14.github.io/boost.http/#_footnotedef_19)] operations. |

CompletionToken

Must fulfill the ASIO requirements for a completion token.

The used handler signature is void(boost::system::error\_code).

4.2.32.3. Parameters

ServerSocket &socket

The socket associated with the *imessage* and *omessage* that will be used for the response.

const Request &imessage

The request message received.

Response &omessage

The message object that should be used to reply the message.

The user might be interested in filling some extra headers here like "content-type" or cache policies.

const filesystem::path &file

The requested file that should be transmitted.

|  |  |
| --- | --- |
| **CAUTION** | If you cannot guarantee the *file* did not change twice during the second covered by the last write time, you should remove all "range" and "if-range" headers from *imessage* before calling this function. It’s possible to construct a more robust file server by making use of system-level APIs that can provide unique identifiers for file revisions. |

bool is\_head\_request

Whether the request was made with a "HEAD" method.

If the received request isn’t "GET" nor "HEAD", you MAY remove all "range" and "if-range" headers and pass the value false to this argument.

|  |  |
| --- | --- |
| **NOTE** | Available only for *overload 2*. |

CompletionToken &&token

The token from which the handler and the return value are extracted.

The extracted handler is called when the operation completes.

4.2.32.4. Return value

Extracted using *token*.

4.2.32.5. See also

* [file\_server\_errc](https://boostgsoc14.github.io/boost.http/#file_server_errc)
* [async\_response\_transmit\_dir](https://boostgsoc14.github.io/boost.http/#async_response_transmit_dir)

4.2.33. async\_response\_transmit\_dir

#include <boost/http/file\_server.hpp>

This function has two overloads.

template<class ServerSocket, class ConvertibleToPath, class Request,

class Response, class CompletionToken>

BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code))

async\_response\_transmit\_dir(ServerSocket &socket,

const ConvertibleToPath &ipath,

const Request &imessage, Response &omessage,

const boost::filesystem::path &root\_dir,

CompletionToken &&token); // (1)

template<class ServerSocket, class ConvertibleToPath, class Request,

class Response, class Predicate, class CompletionToken>

BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code))

async\_response\_transmit\_dir(ServerSocket &socket,

const ConvertibleToPath &ipath,

const Request &imessage, Response &omessage,

const boost::filesystem::path &root\_dir,

Predicate filter, CompletionToken &&token); // (2)

This function does a lot more than just sending bytes. It carries the responsibilities from [async\_response\_transmit\_file](https://boostgsoc14.github.io/boost.http/" \l "async_response_transmit_file), but add a few more of its own:

* It’ll also handle the HTTP method.
* It’ll also handle the file resolution. It does so with respect to the given *root\_dir* argument.

The only feature missing is mime support ("content-type" header). It cannot be done reliably within this abstraction.

This function will handle even the start line and the only acceptable write\_state is empty. It’ll fail on any other write\_state.

All urls are absolute, but are absolute with respect to the given *root\_dir*. This interface provides no means to disable this security check. If the user needs that much complex logic, then it should write its own path resolving solution and use [async\_response\_transmit\_file](https://boostgsoc14.github.io/boost.http/" \l "async_response_transmit_file).

|  |  |
| --- | --- |
| **CAUTION** | This function will call the handler with file\_server\_category::file\_not\_found if the requested file, with respect to the given *root\_dir*, cannot be found. The channel remains untouched in this case, giving the user the opportunity to send custom 404 messages. |
| **CAUTION** | This function will call the handler with file\_server\_category::file\_type\_not\_supported if resolution finishes but this function cannot process the result because the file is not regular (directories, block devices…​). The channel is also left untouched, giving the user the opportunity to use another HTTP consumer. |

4.2.33.1. Template parameters

ServerSocket

Must fulfill the requirements for the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept).

ConvertibleToPath

A type whose instances can be used to construct a boost::filesystem::path object.

Request

Must fulfill the requirements for the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

Response

Must fulfill the requirements for the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

Predicate

A type whose instances are callable and have the following signature:

bool(boost::filesystem::path &resolved\_path)

CompletionToken

Must fulfill the ASIO requirements for a completion token.

The used handler signature is void(boost::system::error\_code).

4.2.33.2. Parameters

ServerSocket &socket

The socket associated with the *imessage* and *omessage* that will be used for the response.

const ConvertibleToPath &ipath

*ipath* (standing for input path) is the **parsed path** from the requested url.

It’s guaranteed that it’ll only be used to construct a boost::filesystem::path object. Thus, the user can fake the requested path to force an *internal redirect*.

|  |  |
| --- | --- |
| **WARNING** | This is the input **path component**, not the input URL. Therefore, you MUST parse the input url before dispatching it to this function and only the path component must be forwarded. Extracting the path is an extra responsibility for the user, but it is an useful abstraction for scenarios where the user doesn’t control the served root dir. Thanks to the security check, this *internal redirect* trick doesn’t work for files outside the *root\_dir*. |

const Request &imessage

The request message received.

|  |  |
| --- | --- |
| **NOTE** | Any request method is acceptable, but any method other than "GET" and "HEAD" will be responded with "405 Method Not Allowed". |

Response &omessage

The message object that should be used to reply the message.

The user might be interested in filling some extra headers here like "content-type" or cache policies.

const boost::filesystem::path &root\_dir

The dir to be interpreted as the root dir of requested files.

Predicate filter

It is applied to the resolved path as the last step before proceeding to file and network operations.

If filter returns false, the functions finishes before touching the channel, with the error\_code file\_server\_category::filter\_set.

|  |  |
| --- | --- |
| **TIP** | It’s possible to use a stateful non-pure *filter* to add response headers and properly process mime types (content-type header). |
| **TIP** | *filter* can also be used to redirect files by modifying the input arg. |

|  |  |
| --- | --- |
| **NOTE** | This function might throw if *filter* throws. In this case, we provide the basic exception guarantee. |
| **NOTE** | Available only for *overload 2*. |

CompletionToken &&token

The token from which the handler and the return value are extracted.

The extracted handler is called when the operation completes.

4.2.33.3. Return value

Extracted using *token*.

4.2.33.4. See also

* [file\_server\_errc](https://boostgsoc14.github.io/boost.http/#file_server_errc)
* [async\_response\_transmit\_file](https://boostgsoc14.github.io/boost.http/#async_response_transmit_file)

4.2.34. read\_state

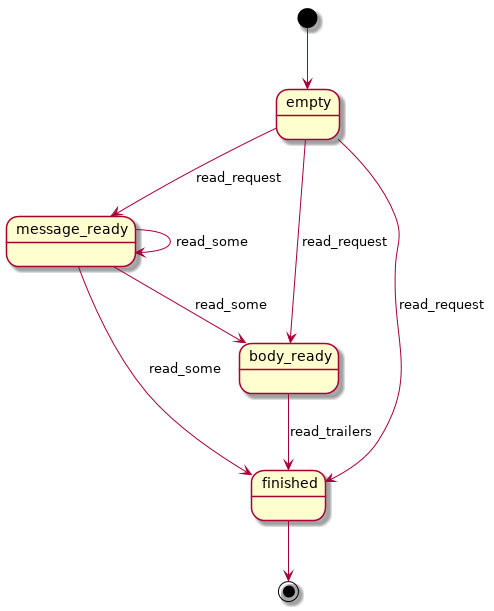
#include <boost/http/read\_state.hpp>

enum class read\_state

Represents the current state in the HTTP incoming request or HTTP incoming response.

Be prepared to face multiple state changes after a single action is scheduled (e.g. you issue read\_message action and the state already changed to finished when the handler is invoked).

4.2.34.1. Member constants (incoming request)



*Figure 2. Incoming request*

empty

This is the initial state. It means that the request object wasn’t read yet.

At this state, you can only issue a read\_request action.

message\_ready

This state is reached from the empty state, once you ask for a new message.

No more read\_request actions can be issued from this state.

From this state, you can issue the read\_some action. The state will change to body\_ready once all body was read. In streaming connections (e.g. HTTP/1.1 chunked entities), this condition (body fully received) might never happen.

Once this state is reached, you can safely use the read start line and the headers.

body\_ready

This state is reached from the message\_ready, once the http producer (e.g. embedded server) fully received the message body.

From this state, you can only issue the read\_trailers action.

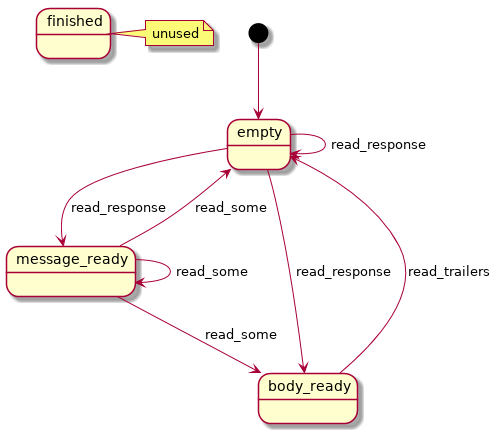
Once this state is reached, you can safely assume that no more body parts will be received.

finished

It means the message is complete and you can no longer issue another read\_request until something else is done (e.g. send another http response). This is a different/special value, because the “something else to do” might not be related to read actions.

It can be reached from body\_ready state, after all trailers have been received. It’s safe to assume that all message data is available at the time this state is reached.

4.2.34.2. Member constants (incoming response)



*Figure 3. Incoming response*

empty

This is the initial state.

There are two ways to interpret this state. It might mean that the response object wasn’t read yet.

Another interpretation is that it was reached from the body\_ready state (directly — through a call to read\_trailers — or indirectly — through a call to read\_some or read\_response), after all trailers have been received. It’s safe to assume that all message data is available if this is the case.

At this state, you can only issue a read\_response action.

message\_ready

This state is reached from the empty state, once you ask for a new message.

No more read\_response actions can be issued from this state.

From this state, you can issue the read\_some action. The state will change to body\_ready once all body was read. In streaming connections (e.g. HTTP/1.1 chunked entities), this condition (body fully received) might never happen.

Once this state is reached, you can safely use the read start line and the headers.

body\_ready

This state is reached from the message\_ready, once the http producer (e.g. an http client) fully received the message body.

From this state, you can only issue the read\_trailers action.

Once this state is reached, you can safely assume that no more body parts will be received.

finished**(UNUSED)**

|  |  |
| --- | --- |
| **NOTE** | Only makes sense in server mode, when reading an incoming request. In client mode, empty target state is used instead. |

4.2.34.3. See also

* [write\_state](https://boostgsoc14.github.io/boost.http/#write_state)
* [Socket](https://boostgsoc14.github.io/boost.http/#socket_concept)

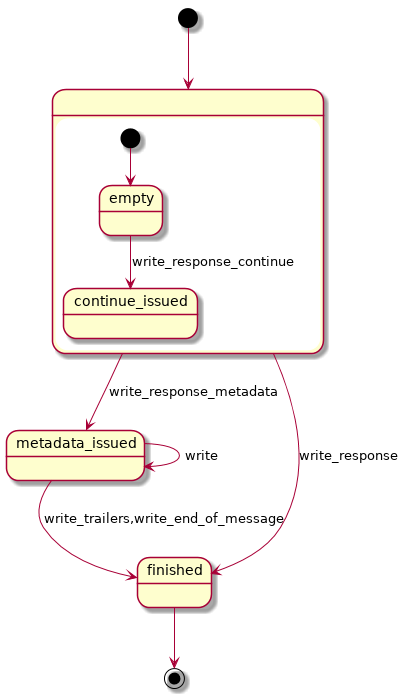
4.2.35. write\_state

#include <boost/http/write\_state.hpp>

enum class write\_state

Represents the current state in the HTTP outgoing response or HTTP outgoing request.

4.2.35.1. Member constants (outgoing response)



*Figure 4. Outgoing response*

empty

This is the initial state.

It means that the response object hasn’t been sent yet.

At this state, you can only issue the metadata or issue a continue action, if continue is supported/used in this HTTP session. Even if continue was requested, issue a continue action is optional and only required if you need the request’s body.

continue\_issued

This state is reached from the empty state, once you issue a continue action.

No more continue actions can be issued from this state.

metadata\_issued

This state can be reached either from empty or continue\_issued.

It happens when the metadata (start line + header section) is issued (through write\_response\_metadata).

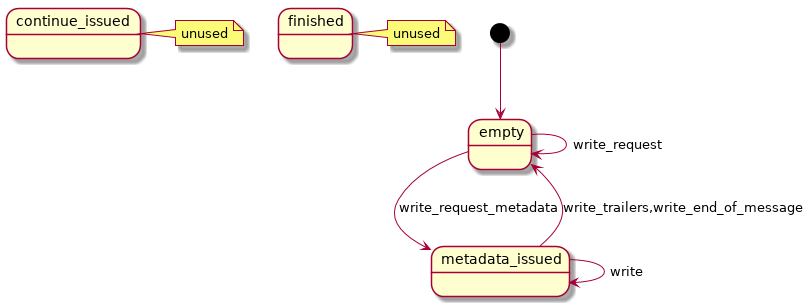
From this state, you can only issue the body, the trailers or the end of the message.

finished

The message is considered complete once this state is reached.

You can no longer issue anything once this state is reached. The underlying channel will change the outgoing\_state to empty once some unspecified event occurs. This event is usually a new request.

4.2.35.2. Member constants (outgoing request)



*Figure 5. Outgoing request*

empty

This is the initial state.

It means that the request object hasn’t been sent yet.

At this state, you can only issue the metadata.

continue\_issued**(UNUSED)**

|  |  |
| --- | --- |
| **NOTE** | Only makes sense in server mode, when sending an outgoing response. |

metadata\_issued

This state can be reached from empty.

It happens when the metadata (start line + header section) is issued (through write\_request\_metadata).

From this state, you can only issue the body, the trailers or the end of the message.

finished**(UNUSED)**

|  |  |
| --- | --- |
| **NOTE** | Only makes sense in server mode, when sending an outgoing response. In client mode, empty target state is used instead. |

4.2.35.3. See also

* [read\_state](https://boostgsoc14.github.io/boost.http/#read_state)
* [Socket](https://boostgsoc14.github.io/boost.http/#socket_concept)

4.2.36. status\_code

#include <boost/http/status\_code.hpp>

enum class status\_code: std::uint\_fast16\_t

This scoped enumeration defines the values for the [“Hypertext Transfer Protocol (HTTP) Status Code Registry”](http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml).

4.2.36.1. Member constants

continue\_request

100

switching\_protocols

101

processing

102

ok

200

created

201

accepted

202

non\_authoritative\_information

203

no\_content

204

reset\_content

205

partial\_content

206

multi\_status

207

already\_reported

208

im\_used

226

multiple\_choices

300

moved\_permanently

301

found

302

see\_other

303

not\_modified

304

use\_proxy

305

switch\_proxy

306

|  |  |
| --- | --- |
| **NOTE** | No longer used. |

temporary\_redirect

307

permanent\_redirect

308

bad\_request

400

unauthorized

401

payment\_required

402

forbidden

403

not\_found

404

method\_not\_allowed

405

not\_acceptable

406

proxy\_authentication\_required

407

request\_timeout

408

conflict

409

gone

410

length\_required

411

precondition\_failed

412

payload\_too\_large

413

uri\_too\_long

414

unsupported\_media\_type

415

requested\_range\_not\_satisfiable

416

expectation\_failed

417

unprocessable\_entity

422

locked

423

failed\_dependency

424

upgrade\_required

426

precondition\_required

428

too\_many\_requests

429

request\_header\_fields\_too\_large

431

internal\_server\_error

500

not\_implemented

501

bad\_gateway

502

service\_unavailable

503

gateway\_timeout

504

http\_version\_not\_supported

505

variant\_also\_negotiates

506

insufficient\_storage

507

loop\_detected

508

not\_extended

510

network\_authentication\_required

511

4.2.36.2. Non-member functions

bool operator==(status\_code lhs, std::uint\_fast16\_t rhs)

Tests if *lhs* and *rhs* are equal.

bool operator==(std::uint\_fast16\_t lhs, status\_code rhs)

Tests if *lhs* and *rhs* are equal.

template<class String> String to\_string(status\_code sc)

Returns the textual representation (i.e. the reason phrase) of *sc*.

4.2.36.3. See also

* [<boost/http/algorithm/write.hpp>](https://boostgsoc14.github.io/boost.http/#write_header)

4.2.37. http\_errc

#include <boost/http/http\_errc.hpp>

enum class http\_errc

This scoped enumeration defines the values for the standard error codes reported by HTTP message producers and consumers. They are intended to be generic and usable by a variety of HTTP producers.

They are designed to work together [http\_category](https://boostgsoc14.github.io/boost.http/" \l "http_errc_http_category).

The traits boost::system::is\_error\_code\_enum and boost::system::is\_error\_condition\_enum are specialized to recognize http\_errc.

4.2.37.1. Member constants

out\_of\_order

Actions issued on the wrong order by the library user.

Make sure to check the examples and the return value from socket.read\_state() and socket.write\_state().

native\_stream\_unsupported

The issued action can only be used when the underlying channel supports native stream, as defined in the [Socket concept page](https://boostgsoc14.github.io/boost.http/#socket_concept).

|  |  |
| --- | --- |
| **TIP** | If you’re using a type fulfilling the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept), you may be interested in the write\_response\_native\_stream() member function. |

parsing\_error

The underlying communication channel sent an invalid message.

buffer\_exhausted

This error should only happen if a poor parser is used.

wrong\_direction

For flexible sockets that select the channel type upon the first use. It happens if you started the socket operations behaving like an HTTP client and later started to behave as an HTTP server, or vice versa, on the same channel.

4.2.37.2. Non-member functions

boost::system::error\_code make\_error\_code(http\_errc e)

Creates an error code using *e* and [http\_category](https://boostgsoc14.github.io/boost.http/" \l "http_errc_http_category).

boost::system::error\_condition make\_error\_condition(http\_errc e)

Creates an error codition using *e* and [http\_category](https://boostgsoc14.github.io/boost.http/" \l "http_errc_http_category).

4.2.37.3. http\_category

const boost::system::error\_category& http\_category()

Obtains a reference to the static error category object for HTTP errors. The object overrides the member function name to return "http" and overrides message to support all values from http\_errc.

4.2.38. file\_server\_errc

#include <boost/http/file\_server.hpp>

enum class file\_server\_errc

This scoped enumeration defines the values for the error codes reported by HTTP file server abstraction shipped with this library.

They’re designed to work with [file\_server\_category](https://boostgsoc14.github.io/boost.http/" \l "file_server_errc_file_server_category).

The traits boost::system::is\_error\_code\_enum and boost::system::is\_error\_condition\_enum are specialized to recognize file\_server\_errc.

4.2.38.1. Member constants

io\_error

When any operation on the file stream fails or throws an exception.

|  |  |
| --- | --- |
| **NOTE** | It’s guaranteed that no operations on the underlying socket were done when this error happens. |

irrecoverable\_io\_error

When any operation on the file stream fails or throws an exception AFTER some operation on the underlying socket already was issued.

write\_state\_not\_supported

If some write operation already was issued before the call to the function that raised this error code.

file\_not\_found

The requested file wasn’t found. The channel is left untouched to give the user the opportunity to send a custom “404 response” or to further forward the request.

file\_type\_not\_supported

The requested file was found but it is not regular (e.g. directories, block devices, links…​). Channel remains untouched.

filter\_set

The user provided filter predicate returned false to cancel the operation. Channel remains untouched.

4.2.38.2. Non-member functions

boost::system::error\_code make\_error\_code(file\_server\_errc e)

Creates an error code using *e* and [file\_server\_category](https://boostgsoc14.github.io/boost.http/" \l "file_server_errc_file_server_category)

boost::system::error\_condition make\_error\_condition(file\_server\_errc e)

Creates an error code using *e* and [file\_server\_category](https://boostgsoc14.github.io/boost.http/" \l "file_server_errc_file_server_category)

4.2.38.3. file\_server\_category

const boost::system::error\_category& file\_server\_category();

Obtains a reference to the static error category object for the file server errors. The object overrides the member function name to return "file\_server" and overrides message to support all values from file\_server\_errc.

4.2.38.4. See also

* [async\_response\_transmit\_file](https://boostgsoc14.github.io/boost.http/#async_response_transmit_file)
* [async\_response\_transmit\_dir](https://boostgsoc14.github.io/boost.http/#async_response_transmit_dir)

4.2.39. Message

A container able to hold generic HTTP messages.

4.2.39.1. Definitions

**HTTP field name**

A string encoded with the *ISO-8859-1* charset whose contents are limited to the chars listed below (case-sensitive):

* A digit (i.e. '0', '1', '2', '3', '4', '5', '6', '7', '8' or '9').
* A lowercase alphabetic (i.e. 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y' or 'z').
* A few special characters: '!', '#', '$', '%', '&', '\'', '\*', '+', '-', '.', '^', '\_', backtick (i.e. '\x60'), '|' or '~'.

|  |  |
| --- | --- |
| **NOTE** | Any uppercase character received through the wire MUST be normalized (i.e. converted to lowercase). |

**HTTP field value**

A string encoded with the *ISO-8859-1* charset whose contents are limited to the chars listed below (whether they’re case-sensitive or not is defined on a header-by-header basis and, as such, they all are considered case-sensitive in this layer of abstraction):

* Any visible *USASCII* character.
* Any character in the closed interval (i.e. both ends are inclusive) between '\x80' and '\xFF'. The use of these characters within the *HTTP field value* is obsolete and should be avoided.
* Space (i.e. '\x20') and horizontal tab (i.e. '\t'). These characters are not allowed in the beginning or in the end of the *HTTP field value*.

**HTTP field**

A pair whose first element is an *HTTP field name* and second element is an *HTTP field value*.

**HTTP header section**

A set of *HTTP fields* received in the same chunk (e.g. the HTTP header section defined in the RFC 7230).

|  |  |
| --- | --- |
| **NOTE** | For fields with equivalent field names, the relative order is preserved. |

4.2.39.2. Notation

X

A type that is a model of Message.

Headers

A type fulfilling the following requirements:

* The C++11 concept of associative containers (*associative.reqmts*) [[20](https://boostgsoc14.github.io/boost.http/#_footnotedef_20)].
* It supports equivalent keys.
* Value type is equal to pair<const Key, T>.
* mapped\_type is available with the same semantics for multimap.
* Headers::key\_type MUST fulfill the requirements for the String concept (i.e. std::basic\_string).

Headers::key\_type::value\_type MUST be able to represent all values in the *ISO-8859-1* charset except for the upper case versions of the alphabetic characters.

|  |  |
| --- | --- |
| **WARNING** | Inserting elements in Headers instances whose keys contains uppercase char(s) invoke undefined behaviour. |

* Headers::mapped\_type MUST fulfill the requirements for the String concept (i.e. std::basic\_string).

Headers::mapped\_type::value\_type MUST be able to represent all values in the *ISO-8859-1* charset.

Body

A type fulfilling the C++ concept of sequence containers (*sequence.reqmts*) whose value\_type can represent byte octets.

a

Object of type X.

ca

Object of type const X.

4.2.39.3. Requirements

| **Expression** | **Return type** | **Precondition** | **Semantics** | **Postcondition** |
| --- | --- | --- | --- | --- |
| std::is\_base\_of<std::true\_type, http::is\_message<X>> | std::true\_type |  |  |  |
| X::headers\_type | Headers |  |  |  |
| X::body\_type | Body |  |  |  |
| a.headers() | X::headers\_type& |  | * It should be used to represent the *HTTP header section* received/to-be-sent before the HTTP body data payload. * Always returns a reference to the same object. * The returned object MUST **NOT** be shared with the one returned by a.trailers(). |  |
| ca.headers() | const X::headers\_type& |  | * It should be used to represent the *HTTP header section* received/to-be-sent before the HTTP body data payload. * Always returns a reference to the same object. * The returned object MUST **NOT** be shared with the one returned by a.trailers(). |  |
| a.body() | X::body\_type& |  | * It should be used to represent chunks of the HTTP body data payload. * Always returns a reference to the same object. |  |
| ca.body() | const X::body\_type& |  | * It should be used to represent chunks of the HTTP body data payload. * Always returns a reference to the same object. |  |
| a.trailers() | X::headers\_type& |  | * It should be used to represent the *HTTP header section* received/to-be-sent after the HTTP body data payload. * Always returns a reference to the same object. * The returned object MUST **NOT** be shared with the one returned by a.headers(). |  |
| ca.trailers() | const X::headers\_type& |  | * It should be used to represent the *HTTP header section* received/to-be-sent after the HTTP body data payload. * Always returns a reference to the same object. * The returned object MUST **NOT** be shared with the one returned by a.headers(). |  |

1. Failing to comply with the “MUST” and “MUST **NOT**” conditions described previously invokes undefined behaviour.

4.2.39.4. See also

* [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).
* [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

4.2.40. Request

A container able to hold HTTP request messages.

4.2.40.1. Refinement of

* [Message](https://boostgsoc14.github.io/boost.http/#message_concept)

4.2.40.2. Notation

X

A type that is a model of Request.

String

A type that is a model of C++'s String.

a

Object of type X.

ca

Object of type const X.

4.2.40.3. Requirements

| **Expression** | **Return type** | **Precondition** | **Semantics** | **Postcondition** |
| --- | --- | --- | --- | --- |
| std::is\_base\_of<std::true\_type, http::is\_request\_message<X>> | std::true\_type |  |  |  |
| X::string\_type | String |  |  |  |
| a.method() | X::string\_type& |  | * It should be used to represent the HTTP request method. * Always returns a reference to the same object. |  |
| ca.method() | const X::string\_type& |  | * It should be used to represent the HTTP request method. * Always returns a reference to the same object. |  |
| a.target() | X::string\_type& |  | * It should be used to represent the HTTP request target. * Always returns a reference to the same object. |  |
| ca.target() | const X::string\_type& |  | * It should be used to represent the HTTP request target. * Always returns a reference to the same object. |  |

4.2.40.4. Models

* [request](https://boostgsoc14.github.io/boost.http/#request)
* [basic\_request](https://boostgsoc14.github.io/boost.http/#basic_request)

4.2.41. Response

A container able to hold HTTP response messages.

4.2.41.1. Refinement of

* [Message](https://boostgsoc14.github.io/boost.http/#message_concept)

4.2.41.2. Notation

X

A type that is a model of Response.

String

A type that is a model of C++'s String.

a

Object of type X.

ca

Object of type const X.

4.2.41.3. Requirements

| **Expression** | **Return type** | **Precondition** | **Semantics** | **Postcondition** |
| --- | --- | --- | --- | --- |
| std::is\_base\_of<std::true\_type, http::is\_response\_message<X>> | std::true\_type |  |  |  |
| X::string\_type | String |  |  |  |
| a.status\_code() | std::uint\_least16\_t& |  | * It should be used to represent the HTTP response status code. * Always returns a reference to the same object. |  |
| ca.status\_code() | const std::uint\_least16\_t& |  | * It should be used to represent the HTTP response status code. * Always returns a reference to the same object. |  |
| a.reason\_phrase() | X::string\_type& |  | * It should be used to represent the HTTP response reason phrase. * Always returns a reference to the same object. |  |
| ca.reason\_phrase() | const X::string\_type& |  | * It should be used to represent the HTTP response reason phrase. * Always returns a reference to the same object. |  |

4.2.41.4. Models

* [response](https://boostgsoc14.github.io/boost.http/#response)
* [basic\_response](https://boostgsoc14.github.io/boost.http/#basic_response)

4.2.42. Socket

Common operations between request and response that the underlying channel should provide.

4.2.42.1. Definitions

**Types fulfilling the**Socket**concept and also possessing the *strict* property**

Instances from this type will **not** insert illegal characters [[21](https://boostgsoc14.github.io/boost.http/#_footnotedef_21)] in the HTTP fields.

**Native stream**

The message size doesn’t need to be know prior to writing the response message. The use of the streaming API can only be used when this property holds. Buffering the body MUST **NOT** be done and the message parts MUST be written as soon as convenient. If this property holds, the user MAY safely use the socket to transmit a live video stream, for instance.

**Message metadata**

The HTTP status line plus the header section.

**Atomic message**

A message which is issued and fully know with a single API call.

Every message is either atomic or chunked.

**Chunked message**

A message which is issued among several API calls.

Every message is either atomic or chunked.

Chunked messages are useful if you don’t know the message in advance (e.g. video streaming). These messages can only be used if native stream is supported.

You always need to be prepared to receive chunked messages and it’s only useful to differ operations which only apply to atomic messages or chunked messages when you’re about to send a message.

4.2.42.2. Notation

Message

A type fulfilling the requirements for the [Message concept](https://boostgsoc14.github.io/boost.http/#message_concept).

m

Object of type Message.

cm

Object of type const Message.

CompletionToken

A type fulfilling the concept of a completion token, as defined in [N4045: Library Foundations for Asynchronous Operations, Revision 2](https://isocpp.org/files/papers/n4045.pdf).

AsyncResultType

BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code))

token

An object of the type CompletionToken.

X

A type that is a model of Socket.

a

Object of type X.

ca

Object of type const X.

4.2.42.3. Requirements

| **Expression** | **Return type** | **Precondition** | **Semantics** | **Postcondition** |
| --- | --- | --- | --- | --- |
| std::is\_base\_of<std::true\_type, http::is\_socket<X>> | std::true\_type |  |  |  |
| X::executor\_type |  |  | The type of the executor associated with the object. |  |
| a.get\_executor() | X::executor\_type |  | Returns the executor associated with the object. |  |
| X::message\_type |  | Socket only supports operations involving a single type of message. | Must be a type fulfilling the [Message concept](https://boostgsoc14.github.io/boost.http/#message_concept). |  |
| ca.is\_open() | bool |  | Determine whether the socket is open. |  |
| ca.read\_state() | [read\_state](https://boostgsoc14.github.io/boost.http/#read_state) |  | Returns the state associated with the Socket meaningful for reading operations. |  |
| ca.write\_state() | [write\_state](https://boostgsoc14.github.io/boost.http/#write_state) |  | Returns the state associated with the Socket meaningful for writing operations. |  |
| a.async\_read\_some(m, token) | AsyncResultType | a.read\_state() == read\_state::message\_ready | Initiate an asynchronous operation to read a part of the message body. Handler is called when at least one byte is read (called with no error set), when the end of message is reached (called with no error set) or when some error occurs.  m.body() and m.trailers() are left in an unspecified state while the operation is in progress. | By the time the handler is called, the part of the message read (if any) is appended to m.body(), if no error happened. |
| a.async\_read\_some(m, token) | AsyncResultType | a.read\_state() != read\_state::message\_ready | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |
| a.async\_read\_trailers(m, token) | AsyncResultType | a.read\_state() == read\_state::body\_ready | Initiate an asynchronous operation to read the trailers. Handler is called when the rest of the message is fully received (called with no error set) or when some error occurs.  m.trailers() is left in an unspecified state while the operation is in progress. | By the time the handler is called, if no error happened, the read trailers (if any) are inserted into m.trailers(). |
| a.async\_read\_trailers(m, token) | AsyncResultType | a.read\_state() != read\_state::body\_ready | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |
| a.async\_write(cm, token) | AsyncResultType | a.write\_state() == write\_state ::metadata\_issued | Initiate an asynchronous operation to write a chunk of the HTTP body data payload (chunked message). Handler is called when the operation completes with an appropriate parameter.  cm.body() MUST **NOT** be modified while the operation is in progress. | By the time the handler is called, the cm.body() data is considered delivered, if no error happened. |
| a.async\_write(cm, token) | AsyncResultType | a.write\_state() != write\_state ::metadata\_issued | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |
| a.async\_write\_trailers(cm, token) | AsyncResultType | a.write\_state() == write\_state ::metadata\_issued | Initiate an asynchronous operation to write the trailer part of the message (chunked message). Handler is called when the operation completes with an appropriate parameter.  cm.trailers() MUST **NOT** be modified while the operation is in progress. | By the time the operation completes, the cm.trailers() data is considered delivered, if no error happened. |
| a.async\_write\_trailers(cm, token) | AsyncResultType | a.write\_state() != write\_state ::metadata\_issued | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |
| a.async\_write\_end\_of\_message (token) | AsyncResultType | a.write\_state() == write\_state ::metadata\_issued | Initiate an asynchronous operation to signalize the sent message is complete (chunked message). Handler is called when the operation completes with an appropriate parameter. | By the time the operation completes, the message is considered complete, if no error happened. |
| a.async\_write\_end\_of\_message (token) | AsyncResultType | a.write\_state() != write\_state ::metadata\_issued | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |

1. Failing to comply with the “MUST” and “MUST **NOT**” conditions described previously invokes undefined behaviour.
2. Any HTTP field name received through the wire is normalized (i.e. uppercase characters are converted to lowercase) before they’re inserted into objects of type Message::headers\_type.
3. The Socket object has the freedom to store information required to further process the incoming message in the user-provided message object. Thus, the library user MUST **NOT** use different message objects in the functions that initiate **read** operations, in the context of the **same** message exchange (i.e. the user can use a different message object to receive a different message). This requirement is extended to refinements of this concept.
4. The Socket object MUST **NOT** insert HTTP headers with empty keys (i.e. "") in message, request or response objects provided by the user.
5. You MUST **NOT** write messages with the "transfer-encoding: chunked" header.
6. You MUST **NOT** write **atomic** messages with the "transfer-encoding" header.

4.2.42.4. Models

* [basic\_socket](https://boostgsoc14.github.io/boost.http/#basic_socket)
* [basic\_buffered\_socket](https://boostgsoc14.github.io/boost.http/#basic_buffered_socket)
* [poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#poly_socket_base)
* [poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#poly_server_socket)
* [poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#poly_client_socket)
* [poly\_socket](https://boostgsoc14.github.io/boost.http/#poly_socket)
* [server\_socket\_adaptor](https://boostgsoc14.github.io/boost.http/#server_socket_adaptor)
* [client\_socket\_adaptor](https://boostgsoc14.github.io/boost.http/#client_socket_adaptor)
* [socket\_adaptor](https://boostgsoc14.github.io/boost.http/#socket_adaptor)

4.2.42.5. See also

* [ServerSocket](https://boostgsoc14.github.io/boost.http/#server_socket_concept)

4.2.43. ServerSocket

Provides operations for HTTP servers.

4.2.43.1. Refinement of

* [Socket](https://boostgsoc14.github.io/boost.http/#socket_concept)

4.2.43.2. Definitions

**Types fulfilling the**ServerSocket**concept and also possessing the *strict* property**

Instances from this type will **not** insert illegal characters [[22](https://boostgsoc14.github.io/boost.http/#_footnotedef_22)] in the HTTP fields.

4.2.43.3. Notation

Request

A type fulfilling the requirements for the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

Response

A type fulfilling the requirements for the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

im

Object of type Request.

om

Object of type const Response.

CompletionToken

A type fulfilling the concept of a completion token, as defined in [N4045: Library Foundations for Asynchronous Operations, Revision 2](https://isocpp.org/files/papers/n4045.pdf).

AsyncResultType

BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code))

token

An object of the type CompletionToken.

X

A type that is a model of ServerSocket.

a

Object of type X.

ca

Object of type const X.

4.2.43.4. Requirements

| **Expression** | **Return type** | **Precondition** | **Semantics** | **Postcondition** |
| --- | --- | --- | --- | --- |
| std::is\_base\_of<std::true\_type, http::is\_server\_socket<X>> | std::true\_type |  |  |  |
| X::request\_type |  | ServerSocket only supports operations involving a single type of request message. | Must be a type fulfilling the requirements for the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept). |  |
| X::response\_type |  | ServerSocket only supports operations involving a single type of response message. | Must be a type fulfilling the requirements for the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept). |  |
| ca.write\_response\_native\_stream() | bool | a.read\_state() != read\_state::empty | Returns whether the current message exchange supports native stream. | The same value and property is maintained until the end of the current message exchange. |
| a.async\_read\_request(im, token) | AsyncResultType | a.read\_state() == read\_state::empty | Initiate an asynchronous operation to read enough of the message to apply the request to a target resource (i.e. request line plus the header section). Handler is called when the operation completes with an appropriate parameter.  im is left in a unspecified state while the operation is in progress.  The ServerSocket object MUST prevent the user from issuing new replies while the request isn’t ready. The prevention MUST be done by changing the write state to write\_state::finished while the read\_request operation is in progress. | By the time the handler is called, im.method() value represents the read method, im.target() represents the read url and all headers for the current request are inserted into m.headers(), if no error happened. |
| a.async\_read\_request(im, token) | AsyncResultType | a.read\_state() != read\_state::empty | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |
| a.async\_write\_response(om, token) | AsyncResultType | a.write\_state() == write\_state::empty || a.write\_state() == write\_state::continue\_issued | Initiate an asynchronous operation to write the response message (atomic message). Handler is called with an appropriate argument when the operation completes.  om MUST **NOT** be modified while the operation is in progress. | * a.write\_state() == write\_state ::finished * a.read\_state() == read\_state ::empty * By the time the handler is called, the om message is considered delivered, if no error happened. |
| a.async\_write\_response(om, token) | AsyncResultType | a.write\_state() != write\_state::empty && a.write\_state() != write\_state::continue\_issued | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |
| a.async\_write\_response\_continue (token) | AsyncResultType | a.write\_state() == write\_state::empty | Initiate an asynchronous operation to write a response with the semantics from a “100 (Continue) response” [[23](https://boostgsoc14.github.io/boost.http/#_footnotedef_23)]. Handler is called when the operation completes with an appropriate parameter. | * a.write\_state() == write\_state ::continue\_issued * By the time the handler is called, the “100 (Continue) response” is considered delivered. |
| a.async\_write\_response\_continue (token) | AsyncResultType | a.write\_state() != write\_state::empty | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |
| a .async\_write\_response\_metadata(om, token) | AsyncResultType | (a.write\_state() == write\_state::empty || a.write\_state() == write\_state::continue\_issued) && a.write\_response\_native\_stream() == true | Initiate an asynchronous operation to write the response metadata (chunked message). Handler is called with an appropriate argument when the operation completes.  om MUST **NOT** be modified while the operation is in progress. | * a.write\_state() == write\_state ::metadata\_issued * By the time the handler is called, the response metadata (i.e. om.status\_code(), om.reason\_phrase() and cm.headers()) is considered delivered, if no error happened. |
| a .async\_write\_response\_metadata(om, token) | AsyncResultType | a.write\_state() != write\_state::empty && a.write\_state() != write\_state::continue\_issued | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order} [[24](https://boostgsoc14.github.io/boost.http/#_footnotedef_24)]. |  |
| a .async\_write\_response\_metadata(om, token) | AsyncResultType | (a.write\_state() == write\_state::empty || a.write\_state() == write\_state::continue\_issued) && a.write\_response\_native\_stream() == false | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc ::native\_stream\_unsupported}. |  |

The following Socket operations are refined with extra semantics/postconditions:

| **Expression** | **Precondition** | **Extra semantics** | **Extra postcondition** |
| --- | --- | --- | --- |
| a.async\_read\_trailers(m, token) | a.read\_state() == read\_state::body\_ready |  | By the time the handler is called, if no error happened, a.read\_state() == http::read\_state::finished. |
| a.async\_write\_trailers(cm, token) | a.write\_state() == write\_state ::metadata\_issued |  | * a.write\_state() == write\_state::finished * a.read\_state() == read\_state ::empty |
| a.async\_write\_end\_of\_message (token) | a.write\_state() == write\_state ::metadata\_issued |  | * a.write\_state() == write\_state::finished * a.read\_state() == read\_state ::empty |

1. Failing to comply with the “MUST” and “MUST **NOT**” conditions described previously invokes undefined behaviour.
2. Any HTTP field name received through the wire is normalized (i.e. uppercase characters are converted to lowercase) before they’re inserted into objects of type Request::headers\_type.
3. If the user pass a "connection: close" header on the message object passed as argument to the async\_write\_response or async\_write\_response\_metadata member-functions, the ServerSocket MUST change the state to closed (i.e. is\_open() will return false).

This behaviour is intended for the communication between the user of this library and the ServerSocket and can differ from the communication between the ServerSocket and the underlying channel.

1. If the ServerSocket reads a message that expects a “100 (Continue) response”, it MUST insert the "expect: 100-continue" header and only one element with the HTTP field name "expect" MUST be present.

This behaviour is intended for the communication between the user of this library and the ServerSocket and can differ from the communication between the ServerSocket and the underlying channel.

1. If the ServerSocket reads a message that does **NOT** expect a “100 (Continue) response”, it MUST erase all the "expect: 100-continue" headers.

This behaviour is intended for the communication between the user of this library and the ServerSocket and can differ from the communication between the ServerSocket and the underlying channel.

1. If the ServerSocket reads a message that represent a desire from the HTTP client to initiate a protocol upgrade, the ServerSocket supports a protocol upgrade and it’ll communicate the client desire to the user of this library, it MUST communicate the desire ensuring all of the following conditions:
   * Ensuring that the "upgrade" (case-insensitive) string is present in the comma-separated list of values from some "connection" header. This rule implictly requires the presence of at least one "connection" header.
   * There is at least one "upgrade" header and all of the "upgrade" headers respect the conditions established in the section 6.7 of the RFC7230.

This behaviour is intended for the communication between the user of this library and the ServerSocket and can differ from the communication between the ServerSocket and the underlying channel.

1. If the ServerSocket isn’t willing to provide a protocol upgrade, then no "upgrade" headers can be present (in other words, all "upgrade" headers MUST be erased before delivering the message to the user of this library).

This behaviour is intended for the communication between the user of this library and the ServerSocket and can differ from the communication between the ServerSocket and the underlying channel.

1. If the "content-length" header is provided to async\_write\_response, then the ServerSocket MUST ignore the message body (i.e. there is no data payload in the reply message) and SHOULD use the user-provided header.

The ServerSocket MUST adopt a behaviour that is compatible with the behaviour defined in the section 3.3.2 of the RFC 7230.

1. The ServerSocket object MUST **NOT** insert HTTP headers with empty keys (i.e. "") in message, request or response objects provided by the user.
2. Informational responses (i.e. 1xx class of status code) indicates an interim response and do not change read\_state or write\_state. Also, these responses carry no body, so all body is ignored (and if the user tries to send such responses using chunks/async\_write\_response\_metadata, error http\_errc::native\_stream\_unsupported will be reported).
3. Some models of ServerSocket might discard informational responses (i.e. 1xx class of status code) that you try to send.

4.2.43.5. Models

* [basic\_socket](https://boostgsoc14.github.io/boost.http/#basic_socket)
* [basic\_buffered\_socket](https://boostgsoc14.github.io/boost.http/#basic_buffered_socket)
* [poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#poly_server_socket)
* [poly\_socket](https://boostgsoc14.github.io/boost.http/#poly_socket)
* [server\_socket\_adaptor](https://boostgsoc14.github.io/boost.http/#server_socket_adaptor)
* [socket\_adaptor](https://boostgsoc14.github.io/boost.http/#socket_adaptor)

4.2.44. ClientSocket

Provides operations for HTTP clients.

4.2.44.1. Refinement of

* [Socket](https://boostgsoc14.github.io/boost.http/#socket_concept)

4.2.44.2. Definitions

**Types fulfilling the**ClientSocket**concept and also possessing the *strict* property**

Instances from this type will **not** insert illegal characters [[25](https://boostgsoc14.github.io/boost.http/#_footnotedef_25)] in the HTTP fields.

4.2.44.3. Notation

Request

A type fulfilling the requirements for the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).

Response

A type fulfilling the requirements for the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).

im

Object of type Response.

om

Object of type const Request.

CompletionToken

A type fulfilling the concept of a completion token, as defined in [N4045: Library Foundations for Asynchronous Operations, Revision 2](https://isocpp.org/files/papers/n4045.pdf).

AsyncResultType

BOOST\_ASIO\_INITFN\_RESULT\_TYPE(CompletionToken, void(boost::system::error\_code))

token

An object of the type CompletionToken.

X

A type that is a model of ClientSocket.

a

Object of type X.

ca

Object of type const X.

4.2.44.4. Requirements

| **Expression** | **Return type** | **Precondition** | **Semantics** | **Postcondition** |
| --- | --- | --- | --- | --- |
| std::is\_base\_of<std::true\_type, http::is\_client\_socket<X>> | std::true\_type |  |  |  |
| X::request\_type |  | ClientSocket only supports operations involving a single type of request message. | Must be a type fulfilling the requirements for the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept). |  |
| X::response\_type |  | ClientSocket only supports operations involving a single type of response message. | Must be a type fulfilling the requirements for the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept). |  |
| a.async\_write\_request(om, token) | AsyncResultType | a.write\_state() == write\_state::empty | Initiate an asynchronous operation to write the request message (atomic message). Handler is called with an appropriate argument when the operation completes.  om MUST **NOT** be modified while the operation is in progress. | By the time the handler is called, the om message is considered delivered, if no error happened. |
| a.async\_write\_request(om, token) | AsyncResultType | a.write\_state() != write\_state::empty | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |
| a .async\_write\_request\_metadata(om, token) | AsyncResultType | a.write\_state() == write\_state::empty | Initiate an asynchronous operation to write the request metadata (chunked message). Handler is called with an appropriate argument when the operation completes.  om MUST **NOT** be modified while the operation is in progress. | * a.write\_state() == write\_state ::metadata\_issued * By the time the handler is called, the request metadata (i.e. om.method(), om.target() and cm.headers()) is considered delivered, if no error happened. |
| a .async\_write\_request\_metadata(om, token) | AsyncResultType | a.write\_state() != write\_state::empty | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |
| a.async\_read\_response(im, token) | AsyncResultType | a.read\_state() == read\_state::empty | Initiate an asynchronous operation to read enough of the message to fill the response metadata (i.e. status line plus the header section). Handler is called when the operation completes with an appropriate parameter.  im is left in a unspecified state while the operation is in progress. | By the time the handler is called, im.status\_code() value represents the read status code, im.reason\_phrase() represents the read reason phrase and all headers for the current response are inserted into im.headers(), if no error happened. |
| a.async\_read\_response(im, token) | AsyncResultType | a.read\_state() != read\_state::empty | No actions are done and the handler from the completion token is called with boost::system::error\_code {http\_errc::out\_of\_order}. |  |

The following Socket operations are refined with extra semantics/postconditions:

| **Expression** | **Precondition** | **Extra semantics** | **Extra postcondition** |
| --- | --- | --- | --- |
| a.async\_read\_trailers(m, token) | a.read\_state() == read\_state::body\_ready |  | By the time the handler is called, if no error happened, a.read\_state() == http::read\_state::empty. |
| a.async\_write\_trailers(cm, token) | a.write\_state() == write\_state ::metadata\_issued |  | a.write\_state() == write\_state::empty |
| a.async\_write\_end\_of\_message (token) | a.write\_state() == write\_state ::metadata\_issued |  | a.write\_state() == write\_state::empty |

1. Failing to comply with the “MUST” and “MUST **NOT**” conditions described previously invokes undefined behaviour.
2. Any HTTP field name received through the wire is normalized (i.e. uppercase characters are converted to lowercase) before they’re inserted into objects of type Response::headers\_type.
3. Upon receiving a message (i.e. async\_read\_response, async\_read\_some or async\_read\_trailers), if connection is gracefully closed in the HTTP-equivalent layer (e.g. "connection: close" header), ClientSocket MUST change the state to closed (i.e. is\_open() will return false).

This behaviour is intended for the communication between the user of this library and the ClientSocket and can differ from the communication between the ClientSocket and the underlying channel.

1. The user communicates the intent to wait for a “100 (Continue) response” by inserting the "expect: 100-continue" header (and more than one element with the HTTP field name "expect" MUST NOT be present in the sent request metadata).

This behaviour is intended for the communication between the user of this library and the ClientSocket and can differ from the communication between the ClientSocket and the underlying channel. For instance, if the ClientSocket doesn’t intend to implement such semantics, it can omit this header from the message sent to the underlying channel and fill a “100 (Continue) response” in the next call the user does to read\_response whether this response was received from the underlying channel or not (i.e. fill a virtual response).

1. If the ClientSocket isn’t willing to provide support for protocol upgrade, then no "upgrade" headers should be sent (in other words, all "upgrade" headers MUST be skipped before delivering the message from the user of this library — or a behaviour that is equivalent in the underlying channel).

This behaviour is intended for the communication between the user of this library and the ClientSocket and can differ from the communication between the ClientSocket and the underlying channel.

1. The ClientSocket object MUST **NOT** insert HTTP headers with empty keys (i.e. "") in message, request or response objects provided by the user.
2. The user of this library MUST **NOT** insert "content-length" or "transfer-encoding" headers in om.

4.2.44.5. Models

* [basic\_socket](https://boostgsoc14.github.io/boost.http/#basic_socket)
* [basic\_buffered\_socket](https://boostgsoc14.github.io/boost.http/#basic_buffered_socket)
* [poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#poly_client_socket)
* [poly\_socket](https://boostgsoc14.github.io/boost.http/#poly_socket)
* [client\_socket\_adaptor](https://boostgsoc14.github.io/boost.http/#client_socket_adaptor)
* [socket\_adaptor](https://boostgsoc14.github.io/boost.http/#socket_adaptor)

4.2.45. <boost/http/algorithm.hpp>

A shorthand to write:

#include <boost/http/algorithm/header.hpp>

#include <boost/http/algorithm/query.hpp>

4.2.45.1. See also

* [<boost/http/algorithm/header.hpp>](https://boostgsoc14.github.io/boost.http/#header_header)
* [<boost/http/algorithm/query.hpp>](https://boostgsoc14.github.io/boost.http/#query_header)

4.2.46. <boost/http/algorithm/header.hpp>

Import the following symbols:

* [header\_to\_ptime](https://boostgsoc14.github.io/boost.http/#header_to_ptime)
* [to\_http\_date](https://boostgsoc14.github.io/boost.http/#to_http_date)
* [header\_value\_all\_of](https://boostgsoc14.github.io/boost.http/#header_value_all_of)
* [header\_value\_any\_of](https://boostgsoc14.github.io/boost.http/#header_value_any_of)
* [header\_value\_none\_of](https://boostgsoc14.github.io/boost.http/#header_value_none_of)
* [header\_value\_for\_each](https://boostgsoc14.github.io/boost.http/#header_value_for_each)
* [etag\_match\_strong](https://boostgsoc14.github.io/boost.http/#etag_match_strong)
* [etag\_match\_weak](https://boostgsoc14.github.io/boost.http/#etag_match_weak)

4.2.47. <boost/http/algorithm/query.hpp>

Import the following symbols:

* [request\_continue\_required](https://boostgsoc14.github.io/boost.http/#request_continue_required)
* [request\_upgrade\_desired](https://boostgsoc14.github.io/boost.http/#request_upgrade_desired)

4.2.48. <boost/http/file\_server.hpp>

Import the following symbols:

* [async\_response\_transmit\_file](https://boostgsoc14.github.io/boost.http/#async_response_transmit_file)
* [async\_response\_transmit\_dir](https://boostgsoc14.github.io/boost.http/#async_response_transmit_dir)
* [file\_server\_errc](https://boostgsoc14.github.io/boost.http/#file_server_errc)

4.2.49. <boost/http/headers.hpp>

Import the following symbol:

* [headers](https://boostgsoc14.github.io/boost.http/#headers)

4.2.50. <boost/http/http\_category.hpp>

Import the http\_category symbol documented at [http\_errc page](https://boostgsoc14.github.io/boost.http/" \l "http_errc_http_category).

4.2.51. <boost/http/http\_errc.hpp>

Import all the symbols documented at the [http\_errc page](https://boostgsoc14.github.io/boost.http/" \l "http_errc).

4.2.52. <boost/http/request.hpp>

Import the following symbols:

* [basic\_request](https://boostgsoc14.github.io/boost.http/#basic_request)
* [request](https://boostgsoc14.github.io/boost.http/#request)
* [headers](https://boostgsoc14.github.io/boost.http/#headers)

4.2.53. <boost/http/response.hpp>

Import the following symbols:

* [basic\_response](https://boostgsoc14.github.io/boost.http/#basic_response)
* [response](https://boostgsoc14.github.io/boost.http/#response)
* [headers](https://boostgsoc14.github.io/boost.http/#headers)

4.2.54. <boost/http/request\_response\_wrapper.hpp>

Import the following symbols:

* [request\_response\_wrapper](https://boostgsoc14.github.io/boost.http/#request_response_wrapper)

4.2.55. <boost/http/poly\_socket.hpp>

Import the following symbols:

* [basic\_poly\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_socket)
* [poly\_socket](https://boostgsoc14.github.io/boost.http/#poly_socket)
* [basic\_poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_server_socket)
* [poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#poly_server_socket)
* [basic\_poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_client_socket)
* [poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#poly_client_socket)
* [poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#poly_socket_base)

4.2.56. <boost/http/poly\_server\_socket.hpp>

Import the following symbols:

* [basic\_poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_server_socket)
* [poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#poly_server_socket)
* [poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#poly_socket_base)

4.2.57. <boost/http/poly\_client\_socket.hpp>

Import the following symbols:

* [basic\_poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#basic_poly_client_socket)
* [poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#poly_client_socket)
* [poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#poly_socket_base)

4.2.58. <boost/http/poly\_socket\_base.hpp>

Import the following symbols:

* [basic\_poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#basic_poly_socket_base)
* [poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#poly_socket_base)

4.2.59. <boost/http/read\_state.hpp>

Import the following symbol:

* [read\_state](https://boostgsoc14.github.io/boost.http/#read_state)

4.2.60. <boost/http/server\_socket\_adaptor.hpp>

Import the following symbols:

* [server\_socket\_adaptor](https://boostgsoc14.github.io/boost.http/#server_socket_adaptor)
* [poly\_server\_socket](https://boostgsoc14.github.io/boost.http/#poly_server_socket)
* [poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#poly_socket_base)

4.2.61. <boost/http/client\_socket\_adaptor.hpp>

Import the following symbols:

* [client\_socket\_adaptor](https://boostgsoc14.github.io/boost.http/#client_socket_adaptor)
* [poly\_client\_socket](https://boostgsoc14.github.io/boost.http/#poly_client_socket)
* [poly\_socket\_base](https://boostgsoc14.github.io/boost.http/#poly_socket_base)

4.2.62. <boost/http/socket\_adaptor.hpp>

Import the following symbols:

* [socket\_adaptor](https://boostgsoc14.github.io/boost.http/#socket_adaptor)
* [poly\_socket](https://boostgsoc14.github.io/boost.http/#poly_socket)

4.2.63. <boost/http/socket.hpp>

Import the following symbols:

* [basic\_socket](https://boostgsoc14.github.io/boost.http/#basic_socket)
* [socket](https://boostgsoc14.github.io/boost.http/#socket)
* [read\_state](https://boostgsoc14.github.io/boost.http/#read_state)
* [write\_state](https://boostgsoc14.github.io/boost.http/#write_state)
* [http\_errc](https://boostgsoc14.github.io/boost.http/#http_errc)

4.2.64. <boost/http/buffered\_socket.hpp>

Import the following symbols:

* [basic\_buffered\_socket](https://boostgsoc14.github.io/boost.http/#basic_buffered_socket)
* [buffered\_socket](https://boostgsoc14.github.io/boost.http/#buffered_socket)
* [read\_state](https://boostgsoc14.github.io/boost.http/#read_state)
* [write\_state](https://boostgsoc14.github.io/boost.http/#write_state)
* [http\_errc](https://boostgsoc14.github.io/boost.http/#http_errc)

4.2.65. <boost/http/status\_code.hpp>

Import the following symbol:

* [status\_code](https://boostgsoc14.github.io/boost.http/#status_code)

4.2.66. <boost/http/write\_state.hpp>

Import the following symbol:

* [write\_state](https://boostgsoc14.github.io/boost.http/#write_state)

4.2.67. <boost/http/traits.hpp>

Import the following symbols:

* [is\_message](https://boostgsoc14.github.io/boost.http/#is_message)
* [is\_request\_message](https://boostgsoc14.github.io/boost.http/#is_request_message)
* [is\_response\_message](https://boostgsoc14.github.io/boost.http/#is_response_message)
* [is\_socket](https://boostgsoc14.github.io/boost.http/#is_socket)
* [is\_server\_socket](https://boostgsoc14.github.io/boost.http/#is_server_socket)
* [is\_client\_socket](https://boostgsoc14.github.io/boost.http/#is_client_socket)

4.2.68. <boost/http/basic\_router.hpp>

Import the following symbols:

* [basic\_router](https://boostgsoc14.github.io/boost.http/#basic_router)

4.2.69. <boost/http/regex\_router.hpp>

Import the following symbols:

* [regex\_router](https://boostgsoc14.github.io/boost.http/#regex_router)

4.2.70. is\_message

#include <boost/http/traits.hpp>

If T is an object fulfilling the [Message concept](https://boostgsoc14.github.io/boost.http/#message_concept), this template inherits std::true\_type. For any other type, this template inherits std::false\_type.

This template may be specialized for a user-defined type to indicate that the type is eligible for operations involving [Message objects](https://boostgsoc14.github.io/boost.http/#message_concept).

Initially, it was considered to create a trait that would automatically detect if T is fullfilling the [Message concept](https://boostgsoc14.github.io/boost.http/#message_concept), but the idea was abandoned, because the [Message concept](https://boostgsoc14.github.io/boost.http/#message_concept) includes behaviour that can only be detected at runtime.

The default definition follows:

template<class T>

struct is\_message

: public std::integral\_constant<bool,

is\_request\_message<T>::value

|| is\_response\_message<T>::value>

{};

4.2.70.1. Template parameters

T

The type to query.

4.2.70.2. See also

* [Message concept](https://boostgsoc14.github.io/boost.http/#message_concept).
* [is\_request\_message](https://boostgsoc14.github.io/boost.http/#is_request_message).
* [is\_response\_message](https://boostgsoc14.github.io/boost.http/#is_response_message).

4.2.71. is\_request\_message

#include <boost/http/traits.hpp>

If T is an object fulfilling the [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept), this template inherits std::true\_type. For any other type, this template inherits std::false\_type.

This template may be specialized for a user-defined type to indicate that the type is eligible for operations involving [Request objects](https://boostgsoc14.github.io/boost.http/#request_concept).

4.2.71.1. Template parameters

T

The type to query.

4.2.71.2. See also

* [Request concept](https://boostgsoc14.github.io/boost.http/#request_concept).
* [is\_message](https://boostgsoc14.github.io/boost.http/#is_message).
* [is\_response\_message](https://boostgsoc14.github.io/boost.http/#is_response_message).

4.2.72. is\_response\_message

#include <boost/http/traits.hpp>

If T is an object fulfilling the [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept), this template inherits std::true\_type. For any other type, this template inherits std::false\_type.

This template may be specialized for a user-defined type to indicate that the type is eligible for operations involving [Response objects](https://boostgsoc14.github.io/boost.http/#response_concept).

4.2.72.1. Template parameters

T

The type to query.

4.2.72.2. See also

* [Response concept](https://boostgsoc14.github.io/boost.http/#response_concept).
* [is\_message](https://boostgsoc14.github.io/boost.http/#is_message).
* [is\_request\_message](https://boostgsoc14.github.io/boost.http/#is_request_message).

4.2.73. is\_socket

#include <boost/http/traits.hpp>

If T is an object fulfilling the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept), this template inherits std::true\_type. For any other type, this template inherits std::false\_type.

This template may be specialized for a user-defined type to indicate that the type is eligible for operations involving [Socket objects](https://boostgsoc14.github.io/boost.http/#socket_concept). If your user-defined type already specializes [is\_server\_socket](https://boostgsoc14.github.io/boost.http/" \l "is_server_socket) or [is\_client\_socket](https://boostgsoc14.github.io/boost.http/" \l "is_client_socket), there is no need to also specialize this template, because this template will, by default, inherit std::true\_type if is\_server\_socket<T>::value || is\_client\_socket<T>::value evaluates to true.

Initially, it was considered to create a trait that would automatically detect if T is fullfilling the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept), but the idea was abandoned, because the [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept) includes behaviour that can only be detected at runtime.

4.2.73.1. Template parameters

T

The type to query.

4.2.73.2. See also

* [Socket concept](https://boostgsoc14.github.io/boost.http/#socket_concept).
* [basic\_socket](https://boostgsoc14.github.io/boost.http/#basic_socket).
* [socket](https://boostgsoc14.github.io/boost.http/#socket).

4.2.74. is\_server\_socket

#include <boost/http/traits.hpp>

If T is an object fulfilling the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept), this template inherits std::true\_type. For any other type, this template inherits std::false\_type.

This template may be specialized for a user-defined type to indicate that the type is eligible for operations involving [ServerSocket objects](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept).

Initially, it was considered to create a trait that would automatically detect if T is fullfilling the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept), but the idea was abandoned, because the [ServerSocket concept](https://boostgsoc14.github.io/boost.http/" \l "server_socket_concept) includes behaviour that can only be detected at runtime.

4.2.74.1. Template parameters

T

The type to query.

4.2.74.2. See also

* [ServerSocket concept](https://boostgsoc14.github.io/boost.http/#server_socket_concept).
* [basic\_socket](https://boostgsoc14.github.io/boost.http/#basic_socket).
* [socket](https://boostgsoc14.github.io/boost.http/#socket).

4.2.75. is\_client\_socket

#include <boost/http/traits.hpp>

If T is an object fulfilling the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept), this template inherits std::true\_type. For any other type, this template inherits std::false\_type.

This template may be specialized for a user-defined type to indicate that the type is eligible for operations involving [ClientSocket objects](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept).

Initially, it was considered to create a trait that would automatically detect if T is fullfilling the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept), but the idea was abandoned, because the [ClientSocket concept](https://boostgsoc14.github.io/boost.http/" \l "client_socket_concept) includes behaviour that can only be detected at runtime.

4.2.75.1. Template parameters

T

The type to query.

4.2.75.2. See also

* [ClientSocket concept](https://boostgsoc14.github.io/boost.http/#client_socket_concept).
* [basic\_socket](https://boostgsoc14.github.io/boost.http/#basic_socket).
* [socket](https://boostgsoc14.github.io/boost.http/#socket).

4.2.76. basic\_router

#include <boost/http/basic\_router.hpp>

Router based on simple test functions. It is implemented as a vector of pairs<test,router>. Where test is a functor that tests a specific path received by a connection, and if returns ture, calls the router function with an arbitary number of arguments.

A test functor could be as simple as bool test\_path(string path) { return path.empty(); }.

The router is an arbitary function, that must match the call to the router itself. So if we call the route as basic\_router(path, arg1, arg2) the router function will be called with router(arg1, arg2).

4.2.76.1. Template parameters

unary\_predicate

Functor that recieves std::string path as an argument and returns true if this route should be selected.

route\_function\_type

Functor of the route destination function.

typename…​ arguments

List of argument type to be passed onto the route destination function.

4.2.76.2. See also

* [regex\_router](https://boostgsoc14.github.io/boost.http/#regex_router).

4.2.77. regex\_router

#include <boost/http/regex\_router.hpp>

Router based on regular expressions. It is implemented as a vector of pairs<regex,router>. Where regex is a std::regex that tests a specific path received by a connection, and if matches, calls the router function with an arbitary number of arguments.

The router is an arbitary function, that must match the call to the router itself. So if we call the route as basic\_router(path, arg1, arg2) the router function will be called with router(arg1, arg2).

4.2.77.1. Template parameters

route\_function\_type

Functor of the route destination function.

typename…​ arguments

List of argument type to be passed onto the route destination function.

4.2.77.2. See also

* [basic\_router](https://boostgsoc14.github.io/boost.http/#basic_router).

4.2.78. token::code::value

#include <boost/http/token.hpp>

namespace token {

struct code

{

enum value

{

error\_insufficient\_data,

error\_set\_method,

error\_use\_another\_connection,

error\_invalid\_data,

error\_no\_host,

error\_invalid\_content\_length,

error\_content\_length\_overflow,

error\_invalid\_transfer\_encoding,

error\_chunk\_size\_overflow,

skip,

method,

request\_target,

version,

status\_code,

reason\_phrase,

field\_name,

field\_value,

end\_of\_headers,

body\_chunk,

end\_of\_body,

trailer\_name,

trailer\_value,

end\_of\_message

};

};

} // namespace token

error\_insufficient\_data

token\_size() of this token will always be zero.

4.2.79. token::symbol::value

#include <boost/http/token.hpp>

namespace token {

struct symbol

{

enum value

{

error,

skip,

method,

request\_target,

version,

status\_code,

reason\_phrase,

field\_name,

field\_value,

end\_of\_headers,

body\_chunk,

end\_of\_body,

trailer\_name,

trailer\_value,

end\_of\_message

};

static value convert(code::value);

};

} // namespace token

4.2.80. token::category::value

#include <boost/http/token.hpp>

namespace token {

struct category

{

enum value

{

status,

data,

structural

};

static value convert(code::value);

static value convert(symbol::value);

};

} // namespace token

4.2.81. token::skip

#include <boost/http/token.hpp>

namespace token {

struct skip

{

static const token::code::value code = token::code::skip;

};

} // namespace token

Used to skip unneeded bytes so user can keep buffer small when asking for more data.

4.2.82. token::field\_name

#include <boost/http/token.hpp>

namespace token {

struct field\_name

{

typedef boost::string\_view type;

static const token::code::value code = token::code::field\_name;

};

} // namespace token

4.2.83. token::field\_value

#include <boost/http/token.hpp>

namespace token {

struct field\_value

{

typedef boost::string\_view type;

static const token::code::value code = token::code::field\_value;

};

} // namespace token

4.2.84. token::body\_chunk

#include <boost/http/token.hpp>

namespace token {

struct body\_chunk

{

typedef asio::const\_buffer type;

static const token::code::value code = token::code::body\_chunk;

};

} // namespace token

4.2.85. token::end\_of\_headers

#include <boost/http/token.hpp>

namespace token {

struct end\_of\_headers

{

static const token::code::value code = token::code::end\_of\_headers;

};

} // namespace token

4.2.86. token::end\_of\_body

#include <boost/http/token.hpp>

namespace token {

struct end\_of\_body

{

static const token::code::value code = token::code::end\_of\_body;

};

} // namespace token

4.2.87. token::trailer\_name

#include <boost/http/token.hpp>

namespace token {

struct trailer\_name

{

typedef boost::string\_view type;

static const token::code::value code = token::code::trailer\_name;

};

} // namespace token

|  |  |
| --- | --- |
| **NOTE** | This token is “implicitly convertible” to field\_name, so to speak. In other words, you can treat it as field\_name at value extraction time (i.e. the reader::{request,response}::value<T>() function). |

4.2.88. token::trailer\_value

#include <boost/http/token.hpp>

namespace token {

struct trailer\_value

{

typedef boost::string\_view type;

static const token::code::value code = token::code::trailer\_value;

};

} // namespace token

|  |  |
| --- | --- |
| **NOTE** | This token is “implicitly convertible” to field\_value, so to speak. In other words, you can treat it as field\_value at value extraction time (i.e. the reader::{request,response}::value<T>() function). |

4.2.89. token::end\_of\_message

#include <boost/http/token.hpp>

namespace token {

struct end\_of\_message

{

static const token::code::value code = token::code::end\_of\_message;

};

} // namespace token

4.2.90. token::method

#include <boost/http/token.hpp>

namespace token {

struct method

{

typedef boost::string\_view type;

static const token::code::value code = token::code::method;

};

} // namespace token

4.2.91. token::request\_target

#include <boost/http/token.hpp>

namespace token {

struct request\_target

{

typedef boost::string\_view type;

static const token::code::value code = token::code::request\_target;

};

} // namespace token

4.2.92. token::version

#include <boost/http/token.hpp>

namespace token {

struct version

{

typedef int type;

static const token::code::value code = token::code::version;

};

} // namespace token

4.2.93. token::status\_code

#include <boost/http/token.hpp>

namespace token {

struct status\_code

{

typedef uint\_least16\_t type;

static const token::code::value code = token::code::status\_code;

};

} // namespace token

4.2.94. token::reason\_phrase

#include <boost/http/token.hpp>

namespace token {

struct reason\_phrase

{

typedef boost::string\_view type;

static const token::code::value code = token::code::reason\_phrase;

};

} // namespace token

4.2.95. reader::request

#include <boost/http/reader/request.hpp>

This class represents an HTTP/1.1 (and HTTP/1.0) incremental parser. It’ll use the token definitions found in [token::code::value](https://boostgsoc14.github.io/boost.http/#token_code_value). You may want to check the [basic parsing tutorial](https://boostgsoc14.github.io/boost.http/#parsing_tutorial1) to learn the basics.

|  |  |
| --- | --- |
| **IMPORTANT** | Once the parser enters in an error state (**and** the error is different than token::code::error\_insufficient\_data), the internal buffer is said to be in an invalidated state. Therefore, the parser won’t access the data anymore and the user is free to invalidate the data (e.g. resize/free it) without calling set\_buffer() or reset() first.  If you want to reuse the same reader object to parse another stream, just call reset(). |

4.2.95.1. Member types

typedef std::size\_t size\_type

Type used to represent sizes.

typedef const char value\_type

Type used to represent the value of a single element in the buffer.

typedef value\_type \*pointer

Pointer-to-value type.

typedef boost::string\_view view\_type

Type used to refer to non-owning string slices.

4.2.95.2. Member functions

request()

Constructor.

void reset()

After a call to this function, the object has the same internal state as an object that was just constructed.

token::code::value code() const

Use it to inspect current token. Returns code.

|  |  |
| --- | --- |
| **NOTE** | The following values are **never** returned:   * token::code::error\_set\_method. * token::code::error\_use\_another\_connection. * token::code::status\_code. * token::code::reason\_phrase. |

token::symbol::value symbol() const

Use it to inspect current token. Returns symbol.

|  |  |
| --- | --- |
| **NOTE** | The following values are **never** returned:   * token::symbol::status\_code. * token::symbol::reason\_phrase. |

token::category::value category() const

Use it to inspect current token. Returns category.

size\_type token\_size() const

Returns the size of current token.

|  |  |  |
| --- | --- | --- |
| **NOTE** | After you call next(), you’re free to remove, from the buffer, the amount of bytes equals to the value returned here.  If you do remove the parsed data from the buffer, the address of the data shouldn’t change (i.e. you must not invalidate the pointers/iterators to old unparsed data). If you do change the address of old unparsed data, call set\_buffer before using this object again.  *Example*  std::size\_t nparsed = reader.token\_size();  reader.next();  buffer.erase(0, nparsed);  reader.set\_buffer(buffer); | |
| **WARNING** | | Do **not** use string\_length(reader.value<T>()) to compute the token size. string\_length(reader.value<T>()) and reader.token\_size() may differ. [Check the advanced parsing tutorial for more details](https://boostgsoc14.github.io/boost.http/#parsing_tutorial2). |

template<class T> typename T::type value() const

Extracts the value of current token and returns it.

T must be one of:

* token::method.
* token::request\_target.
* token::version.
* token::field\_name.
* token::field\_value.
* token::body\_chunk.

|  |  |  |
| --- | --- | --- |
| **WARNING** | | The assert(code() == T::code) precondition is assumed. |
| **NOTE** | This parser doesn’t buffer data. The value is extracted directly from buffer. | |

token::code::value expected\_token() const

Returns the expected token code.

Useful when the buffer has been exhausted and code() == token::code::error\_insufficient\_data. Use it to respond with “URL/HTTP-header/…​ too long” or another error-handling strategy.

|  |  |
| --- | --- |
| **WARNING** | The returned value is a **heuristic**, not a *truth*. If your buffer is too small, the buffer will be exhausted with too little info to know which element is expected for sure.  For instance, expected\_token() might return token::code::field\_name, but when you have enough info in the buffer, the actual token happens to be token::code::end\_of\_headers. |

void next()

Consumes the current token and advances in the buffer.

|  |  |
| --- | --- |
| **NOTE** | Given the current token is complete (i.e. code() != token::code::error\_insufficient\_data), a call to this function **always** consumes the current token. |

void set\_buffer(asio::const\_buffer inbuffer)

Sets buffer to *inbuffer*.

|  |  |  |
| --- | --- | --- |
| **NOTE** | *inbuffer* should hold the data at the same point of unparsed data from the internal buffer from before this call.  *Example*  std::size\_t nparsed = reader.token\_size();  // now unparsed data becomes ahead  // of `buffer.begin()`  reader.next();  reader.set\_buffer(buffer + nparsed); | |
| **WARNING** | | The reader object follows the HTTP stream orchestrated by the continuous flow of set\_buffer() and next(). You should treat this region as read-only. For instance, if I pass "header-a: something" to the reader and then change the contents to "header-a: another thing", there are no guarantees about the reader object behaviour. You can safely change only the contents of the buffer region not yet exposed to reader through reader.set\_buffer(some\_buffer) (i.e. the region outside of some\_buffer never seen by reader). |

|  |  |
| --- | --- |
| **NOTE** | You’re free to pass larger buffers at will.  You’re also free to pass a buffer just as big as current token (i.e. token\_size()). In other words, you’re free to shrink the buffer **if** the new buffer is at least as big as current token. |
| **TIP** | If you want to free the buffer while maintaining the reader object valid, just set the buffer to current token size, call next() and then set buffer to an empty buffer.  Do notice that this will consume current token as well. And as values are decoded directly from the buffer, this strategy is the only choice.  *Example*  reader.set\_buffer(boost::asio::buffer(buffer, reader.token\_size()));  reader.next();  reader.set\_buffer(boost::asio::const\_buffer());  buffer.clear(); |

size\_type parsed\_count() const

Returns the number of bytes parsed **since**set\_buffer**was last called**.

|  |  |
| --- | --- |
| **TIP** | You can use it to go away with the nparsed variable shown in the [principles on parsing tutorial](https://boostgsoc14.github.io/boost.http/#parsing_tutorial1). I’m sorry about the “*you must keep track of the number of discarded bytes*” lie I told you before, but as [one great explainer once told](https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Scott-Meyers-Universal-References-in-Cpp11):  *As I look upon you…​ it occurs to me that you may not have the necessary level of maturity to handle the truth.*  *— Scott MeyersC++ and Beyond 2012: Universal References in C++11*  That lie was useful to explain some core concepts behind this library. |

4.2.95.3. See also

* [What are the differences between reader::request and reader::response?](https://boostgsoc14.github.io/boost.http/#request_response_diff)

4.2.96. reader::response

#include <boost/http/reader/response.hpp>

This class represents an HTTP/1.1 (and HTTP/1.0) incremental parser. It’ll use the token definitions found in [token::code::value](https://boostgsoc14.github.io/boost.http/#token_code_value). You may want to check the [basic parsing tutorial](https://boostgsoc14.github.io/boost.http/#parsing_tutorial1) to learn the basics.

|  |  |
| --- | --- |
| **IMPORTANT** | Once the parser enters in an error state (**and** the error is different than token::code::error\_insufficient\_data), the internal buffer is said to be in an invalidated state. Therefore, the parser won’t access the data anymore and the user is free to invalidate the data (e.g. resize/free it) without calling set\_buffer() or reset() first.  If you want to reuse the same reader object to parse another stream, just call reset(). |

4.2.96.1. Member types

typedef std::size\_t size\_type

Type used to represent sizes.

typedef const char value\_type

Type used to represent the value of a single element in the buffer.

typedef value\_type \*pointer

Pointer-to-value type.

typedef boost::string\_view view\_type

Type used to refer to non-owning string slices.

4.2.96.2. Member functions

response()

Constructor.

void set\_method(view\_type method)

Use it to inform the request method of the request message associated with this response message. This is necessary internally to compute the body size. If you do not call this function when code() == token::code::status\_code, then token::code::error\_set\_method will be the next token.

|  |  |
| --- | --- |
| **WARNING** | The assert(code() == token::code::status\_code) precondition is assumed. |

void reset()

After a call to this function, the object has the same internal state as an object that was just constructed.

void puteof()

If the connection is closed, call this function. HTTP/1.0 used this event to signalize token::code::end\_of\_body.

token::code::value code() const

Use it to inspect current token. Returns code.

|  |  |
| --- | --- |
| **NOTE** | The following values are **never** returned:   * token::code::error\_no\_host. * token::code::method. * token::code::request\_target. |

token::symbol::value symbol() const

Use it to inspect current token. Returns symbol.

|  |  |
| --- | --- |
| **NOTE** | The following values are **never** returned:   * token::symbol::method. * token::symbol::request\_target. |

token::category::value category() const

Use it to inspect current token. Returns category.

size\_type token\_size() const

Returns the size of current token.

|  |  |  |
| --- | --- | --- |
| **NOTE** | After you call next(), you’re free to remove, from the buffer, the amount of bytes equals to the value returned here.  If you do remove the parsed data from the buffer, the address of the data shouldn’t change (i.e. you must not invalidate the pointers/iterators to old unparsed data). If you do change the address of old unparsed data, call set\_buffer before using this object again.  *Example*  std::size\_t nparsed = reader.token\_size();  reader.next();  buffer.erase(0, nparsed);  reader.set\_buffer(buffer); | |
| **WARNING** | | Do **not** use string\_length(reader.value<T>()) to compute the token size. string\_length(reader.value<T>()) and reader.token\_size() may differ. [Check the advanced parsing tutorial for more details](https://boostgsoc14.github.io/boost.http/#parsing_tutorial2). |

template<class T> typename T::type value() const

Extracts the value of current token and returns it.

T must be one of:

* token::status\_code.
* token::version.
* token::reason\_phrase.
* token::field\_name.
* token::field\_value.
* token::body\_chunk.

|  |  |  |
| --- | --- | --- |
| **WARNING** | | The assert(code() == T::code) precondition is assumed. |
| **NOTE** | This parser doesn’t buffer data. The value is extracted directly from buffer. | |

token::code::value expected\_token() const

Returns the expected token code.

Useful when the buffer has been exhausted and code() == token::code::error\_insufficient\_data. Use it to log error to *cout* or another error-handling strategy.

|  |  |
| --- | --- |
| **WARNING** | The returned value is a **heuristic**, not a *truth*. If your buffer is too small, the buffer will be exhausted with too little info to know which element is expected for sure.  For instance, expected\_token() might return token::code::field\_name, but when you have enough info in the buffer, the actual token happens to be token::code::end\_of\_headers. |

void next()

Consumes the current token and advances in the buffer.

|  |  |
| --- | --- |
| **NOTE** | Given the current token is complete (i.e. code() != token::code::error\_insufficient\_data), a call to this function **always** consumes the current token. |

void set\_buffer(asio::const\_buffer inbuffer)

Sets buffer to *inbuffer*.

|  |  |  |
| --- | --- | --- |
| **NOTE** | *inbuffer* should hold the data at the same point of unparsed data from the internal buffer from before this call.  *Example*  std::size\_t nparsed = reader.token\_size();  // now unparsed data becomes ahead  // of `buffer.begin()`  reader.next();  reader.set\_buffer(buffer + nparsed); | |
| **WARNING** | | The reader object follows the HTTP stream orchestrated by the continuous flow of set\_buffer() and next(). You should treat this region as read-only. For instance, if I pass "header-a: something" to the reader and then change the contents to "header-a: another thing", there are no guarantees about the reader object behaviour. You can safely change only the contents of the buffer region not yet exposed to reader through reader.set\_buffer(some\_buffer) (i.e. the region outside of some\_buffer never seen by reader). |

|  |  |
| --- | --- |
| **NOTE** | You’re free to pass larger buffers at will.  You’re also free to pass a buffer just as big as current token (i.e. token\_size()). In other words, you’re free to shrink the buffer **if** the new buffer is at least as big as current token. |
| **TIP** | If you want to free the buffer while maintaining the reader object valid, just set the buffer to current token size, call next() and then set buffer to an empty buffer.  Do notice that this will consume current token as well. And as values are decoded directly from the buffer, this strategy is the only choice.  *Example*  reader.set\_buffer(boost::asio::buffer(buffer, reader.token\_size()));  reader.next();  reader.set\_buffer(boost::asio::const\_buffer());  buffer.clear(); |

size\_type parsed\_count() const

Returns the number of bytes parsed **since**set\_buffer**was last called**.

|  |  |
| --- | --- |
| **TIP** | You can use it to go away with the nparsed variable shown in the [principles on parsing tutorial](https://boostgsoc14.github.io/boost.http/#parsing_tutorial1). I’m sorry about the “*you must keep track of the number of discarded bytes*” lie I told you before, but as [one great explainer once told](https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Scott-Meyers-Universal-References-in-Cpp11):  *As I look upon you…​ it occurs to me that you may not have the necessary level of maturity to handle the truth.*  *— Scott MeyersC++ and Beyond 2012: Universal References in C++11*  That lie was useful to explain some core concepts behind this library. |

4.2.96.3. See also

* [What are the differences between reader::request and reader::response?](https://boostgsoc14.github.io/boost.http/#request_response_diff)

4.2.97. syntax::chunk\_size

#include <boost/http/syntax/chunk\_size.hpp>

namespace syntax {

template<class CharT>

struct chunk\_size {

typedef basic\_string\_view<CharT> view\_type;

BOOST\_SCOPED\_ENUM\_DECLARE\_BEGIN(result)

{

invalid,

ok,

overflow

}

BOOST\_SCOPED\_ENUM\_DECLARE\_END(result)

static std::size\_t match(view\_type view);

template<class Target>

static result decode(view\_type in, Target &out);

};

} // namespace syntax

4.2.98. syntax::content\_length

#include <boost/http/syntax/content\_length.hpp>

namespace syntax {

template<class CharT>

struct content\_length {

typedef basic\_string\_view<CharT> view\_type;

BOOST\_SCOPED\_ENUM\_DECLARE\_BEGIN(result)

{

invalid,

ok,

overflow

}

BOOST\_SCOPED\_ENUM\_DECLARE\_END(result)

template<class Target>

static result decode(view\_type in, Target &out);

};

} // namespace syntax

4.2.99. syntax::strict\_crlf

#include <boost/http/syntax/crlf.hpp>

namespace syntax {

template<class CharT>

struct strict\_crlf {

typedef basic\_string\_view<CharT> view\_type;

static std::size\_t match(view\_type view);

};

} // namespace syntax

4.2.100. syntax::liberal\_crlf

#include <boost/http/syntax/crlf.hpp>

namespace syntax {

template<class CharT>

struct liberal\_crlf {

typedef basic\_string\_view<CharT> view\_type;

BOOST\_SCOPED\_ENUM\_DECLARE\_BEGIN(result)

{

crlf,

lf,

insufficient\_data,

invalid\_data,

}

BOOST\_SCOPED\_ENUM\_DECLARE\_END(result)

static result match(view\_type view);

};

} // namespace syntax

4.2.101. syntax::field\_name

#include <boost/http/syntax/field\_name.hpp>

namespace syntax {

template<class CharT>

struct field\_name {

typedef basic\_string\_view<CharT> view\_type;

static std::size\_t match(view\_type view);

};

} // namespace syntax

4.2.102. syntax::left\_trimmed\_field\_value

#include <boost/http/syntax/field\_value.hpp>

namespace syntax {

template<class CharT>

struct left\_trimmed\_field\_value {

typedef basic\_string\_view<CharT> view\_type;

static std::size\_t match(view\_type view);

};

} // namespace syntax

4.2.103. syntax::ows

#include <boost/http/syntax/ows.hpp>

namespace syntax {

template<class CharT>

struct ows {

typedef basic\_string\_view<CharT> view\_type;

static std::size\_t match(view\_type view);

};

} // namespace syntax

4.2.104. syntax::reason\_phrase

#include <boost/http/syntax/reason\_phrase.hpp>

namespace syntax {

template<class CharT>

struct reason\_phrase {

typedef basic\_string\_view<CharT> view\_type;

static std::size\_t match(view\_type view);

};

} // namespace syntax

4.2.105. syntax::status\_code

#include <boost/http/syntax/status\_code.hpp>

namespace syntax {

template<class CharT>

struct status\_code {

typedef basic\_string\_view<CharT> view\_type;

static std::size\_t match(view\_type view);

static uint\_least16\_t decode(view\_type view);

};

} // namespace syntax

4.2.106. <boost/http/token.hpp>

Import the following symbols:

* [token::code::value](https://boostgsoc14.github.io/boost.http/#token_code_value)
* [token::symbol::value](https://boostgsoc14.github.io/boost.http/#token_symbol_value)
* [token::category::value](https://boostgsoc14.github.io/boost.http/#token_category_value)
* [token::skip](https://boostgsoc14.github.io/boost.http/#token_skip)
* [token::field\_name](https://boostgsoc14.github.io/boost.http/#token_field_name)
* [token::field\_value](https://boostgsoc14.github.io/boost.http/#token_field_value)
* [token::body\_chunk](https://boostgsoc14.github.io/boost.http/#token_body_chunk)
* [token::end\_of\_headers](https://boostgsoc14.github.io/boost.http/#token_end_of_headers)
* [token::end\_of\_body](https://boostgsoc14.github.io/boost.http/#token_end_of_body)
* [token::end\_of\_message](https://boostgsoc14.github.io/boost.http/#token_end_of_message)
* [token::method](https://boostgsoc14.github.io/boost.http/#token_method)
* [token::request\_target](https://boostgsoc14.github.io/boost.http/#token_request_target)
* [token::version](https://boostgsoc14.github.io/boost.http/#token_version)
* [token::status\_code](https://boostgsoc14.github.io/boost.http/#token_status_code)
* [token::reason\_phrase](https://boostgsoc14.github.io/boost.http/#token_reason_phrase)

4.2.107. <boost/http/reader/request.hpp>

Import the following symbols:

* [reader::request](https://boostgsoc14.github.io/boost.http/#reader_request)

4.2.108. <boost/http/reader/response.hpp>

Import the following symbols:

* [reader::response](https://boostgsoc14.github.io/boost.http/#reader_response)

4.2.109. <boost/http/syntax/chunk\_size.hpp>

Import the following symbols:

* [syntax::chunk\_size](https://boostgsoc14.github.io/boost.http/#syntax_chunk_size)

4.2.110. <boost/http/syntax/content\_length.hpp>

Import the following symbols:

* [syntax::content\_length](https://boostgsoc14.github.io/boost.http/#syntax_content_length)

4.2.111. <boost/http/syntax/crlf.hpp>

Import the following symbols:

* [syntax::strict\_crlf](https://boostgsoc14.github.io/boost.http/#syntax_strict_crlf)
* [syntax::liberal\_crlf](https://boostgsoc14.github.io/boost.http/#syntax_liberal_crlf)

4.2.112. <boost/http/syntax/field\_name.hpp>

Import the following symbols:

* [syntax::field\_name](https://boostgsoc14.github.io/boost.http/#syntax_field_name)

4.2.113. <boost/http/syntax/field\_value.hpp>

Import the following symbols:

* [syntax::left\_trimmed\_field\_value](https://boostgsoc14.github.io/boost.http/#syntax_left_trimmed_field_value)

4.2.114. <boost/http/syntax/ows.hpp>

Import the following symbols:

* [syntax::ows](https://boostgsoc14.github.io/boost.http/#syntax_ows)

4.2.115. <boost/http/syntax/reason\_phrase.hpp>

Import the following symbols:

* [syntax::reason\_phrase](https://boostgsoc14.github.io/boost.http/#syntax_reason_phrase)

4.2.116. <boost/http/syntax/status\_code.hpp>

Import the following symbols:

* [syntax::status\_code](https://boostgsoc14.github.io/boost.http/#syntax_status_code)

[**1**](https://boostgsoc14.github.io/boost.http/#_footnoteref_1). The larger explanation being: incomplete tokens are kept in buffer, so you are not required to allocate them on a secondary buffer, but you’re still allowed to mutate/move/grow the buffer (as opposed to…​ say…​ some C++ iterators that would get invalidated once the container changes).

[**2**](https://boostgsoc14.github.io/boost.http/#_footnoteref_2). parsing is done one token at a time no matter how much data is buffered

[**3**](https://boostgsoc14.github.io/boost.http/#_footnoteref_3). Proper extraction of header fields (some popular parsers like NodeJS’s will force you to know protocol details and parser internals to manually remove leading and trailing whitespace from field values). This point is not only about header field extraction, but really about a parser that indeed abstract and understand the protocol and that is easy to use right by people that don’t know HTTP.

[**4**](https://boostgsoc14.github.io/boost.http/#_footnoteref_4). Useful to avoid the gotchas of ASIO composed operations. See <http://sourceforge.net/p/asio/mailman/message/32259256/> for more information.

[**5**](https://boostgsoc14.github.io/boost.http/#_footnoteref_5). See an example at <http://sourceforge.net/p/axiomq/code/ci/master/tree/include/axiomq/basic_queue_socket.hpp>.

[**6**](https://boostgsoc14.github.io/boost.http/#_footnoteref_6). for tests

[**7**](https://boostgsoc14.github.io/boost.http/#_footnoteref_7). Used in the core library.

[**8**](https://boostgsoc14.github.io/boost.http/#_footnoteref_8). For tests.

[**9**](https://boostgsoc14.github.io/boost.http/#_footnoteref_9). For the examples and tests.

[**10**](https://boostgsoc14.github.io/boost.http/#_footnoteref_10). For the file\_server.

[**11**](https://boostgsoc14.github.io/boost.http/#_footnoteref_11). For the documentation.

[**12**](https://boostgsoc14.github.io/boost.http/#_footnoteref_12). Like [axiomq’s basic\_queue\_socket](https://sourceforge.net/p/axiomq/code/ci/master/tree/include/axiomq/basic_queue_socket.hpp)

[**13**](https://boostgsoc14.github.io/boost.http/#_footnoteref_13). If I continue to develop the Boost.Http’s message framework, that’s the solution that will be adopted to this particular problem.

[**14**](https://boostgsoc14.github.io/boost.http/#_footnoteref_14). Consult Occam’s razor.

[**15**](https://boostgsoc14.github.io/boost.http/#_footnoteref_15). Defined in RFC 7231, section 7.1.1.1.

[**16**](https://boostgsoc14.github.io/boost.http/#_footnoteref_16). Defined in RFC 7232.

[**17**](https://boostgsoc14.github.io/boost.http/#_footnoteref_17). Defined in RFC 7232.

[**18**](https://boostgsoc14.github.io/boost.http/#_footnoteref_18). Defined in RFC 7231, section 5.1.1.

[**19**](https://boostgsoc14.github.io/boost.http/#_footnoteref_19). objects with contiguous storage of bytes.

[**20**](https://boostgsoc14.github.io/boost.http/#_footnoteref_20). The C++11 update gives extra guarantees about preserving the insertion ordering for elements with equivalent keys.

[**21**](https://boostgsoc14.github.io/boost.http/#_footnoteref_21). Defined in the [Message concept’s “definitions” section](https://boostgsoc14.github.io/boost.http/#message_concept).

[**22**](https://boostgsoc14.github.io/boost.http/#_footnoteref_22). Defined in the [Message concept’s “definitions” section](https://boostgsoc14.github.io/boost.http/#message_concept).

[**23**](https://boostgsoc14.github.io/boost.http/#_footnoteref_23). Defined in RFC 7231, section 5.1.1.

[**24**](https://boostgsoc14.github.io/boost.http/#_footnoteref_24). The notification of the error http\_errc::out\_of\_order has priority over http\_errc::native\_stream\_unsupported because these errors present a programming logic error that always will happen (i.e. they aren’t tied to specific runtime behaviour). If they aren’t fixed, the program won’t work either way.

[**25**](https://boostgsoc14.github.io/boost.http/#_footnoteref_25). Defined in the [Message concept’s “definitions” section](https://boostgsoc14.github.io/boost.http/#message_concept).