embedded_http_server — Manual

Version 1.2

A lightweight C++ library for embedding an HTTP/HTTPS server in any application

Table of Contents

Introduction

`embedded_http_server` is a small, self-contained C++ library that lets you add an HTTP (or HTTPS) server directly inside your application — no external web server required. The library is intentionally minimal: it is designed to be compiled straight into your project alongside your own source files.

Typical use cases include:

- Exposing a local browser-based control panel for a desktop application.
- Serving diagnostic or monitoring data from a long-running service.
- Providing a REST-like interface between processes on the same machine.
- Rapid prototyping of web-based user interfaces driven by C++ back-end logic.

Architecture overview

The library has two layers:

`sockets.h / sockets.cpp`
: A thin, portable IOStream-compatible wrapper around TCP sockets (and, on POSIX systems, Unix-domain sockets). You rarely need to use this layer directly.

`http_server.h / http_server.cpp`
: HTTP protocol handling, request routing, and the public API that your application calls.

Each incoming client connection is served by a dedicated `std::thread`, so multiple clients can be handled concurrently. The server is a singleton: `http::server_start()` occupies the calling thread for the lifetime of the server.

The request dispatcher works as follows:

1. If a dynamic handler has been registered for the requested path, it is called.
2. Otherwise, the server looks for a file with the matching path under the configured *base directory* and serves it as a static file.

License

Building the Library

Because `embedded_http_server` is designed to be compiled directly into your project, there is no separate build step. Include the two source files in your own compilation:

```
c++ -std=c++17 -pthread -o myapp \
    myapp.cpp \
    path/to/embedded_http_server-1.2/src/http_server.cpp \
    path/to/embedded_http_server-1.2/src/sockets.cpp \
    -I path/to/embedded_http_server-1.2/src
```

Your application only needs to include `http_server.h`:

```
#include "http_server.h"
```

Requirements:

- A C++17-capable compiler.
- POSIX threads (`-pthread`). On Windows, link with `Ws2_32.lib` instead.

Optional HTTPS support

HTTPS is available when you build with OpenSSL 3. Add `-DSOCKETS_SSL` to the compiler flags and link with `-lssl -lcrypto`:

```
c++ -std=c++17 -pthread -DSOCKETS_SSL -o myapp \
    myapp.cpp \
    path/to/embedded_http_server-1.2/src/http_server.cpp \
    path/to/embedded_http_server-1.2/src/sockets.cpp \
    path/to/embedded_http_server-1.2/src/sockets_ssl.cpp \
    -I path/to/embedded_http_server-1.2/src \
    -lssl -lcrypto
```

The examples directory contains ready-to-use `Makefile` files for each scenario.

Starting the Server

`http::server_start()` is the entry point that starts the server. Call it **last** in `main()` — it blocks indefinitely while the server is running and only returns if a fatal error occurs.

Basic HTTP

```
// Serve static files from the ./files directory on port 12345.
http::server_start(12345, "files");
```

The second argument is the *base directory* from which static files are served. Requests that do not match any dynamic handler are resolved against this directory.

With logging

Pass any `std::ostream` as a third argument to receive diagnostic messages:

```
http::server_start(12345, "files", std::cerr);
```

You can narrow down which messages are printed using the optional bitmask parameter:

```
using namespace http;
http::server_start(12345, "files", std::cerr,
    log_connections | log_dynamic_requests);
```

Available log flags:

Flag	Meaning
`log_connections`	Client connect / disconnect events.
`log_static_requests`	Incoming requests handled by the static file server.
`log_static_responses`	Responses sent for static file requests.
`log_dynamic_requests`	Incoming requests dispatched to a registered handler.
`log_dynamic_responses`	Responses sent by registered handlers.
`log_everything`	All of the above (default when logging is enabled).

HTTPS

Pass an `http::ssl_server_config` (paths to a PEM certificate and private key) as the third argument:

```
http::server_start(12345, "files", {"server.crt", "server.key"});

// With logging:
http::server_start(12345, "files", {"server.crt", "server.key"}, std::cerr);
```

A failed TLS handshake on an individual connection is caught and logged; the server continues running normally.

Registering Dynamic Handlers

Handlers must be registered **before** `server_start()` is called. Each handler is bound to a *name* that maps directly to the URL path `/<name>`.

GET handlers

Three flavours are available depending on how much control you need over the HTTP response.

HTML handler (`register_html_get_action`)

Your function writes only the response body. The server automatically generates correct HTTP headers with `Content-Type: text/html`.

Signature:

```
void handler(std::ostream & out,
             const std::string & path,
             const std::string & params);
```

- `out` — write your HTML content here.
- `path` — the URL path that was requested (e.g. `/my_action`).
- `params` — the raw query string (everything after ?), or empty.

Example — a page that shows the current time:

```
void time_page(std::ostream & out,
    const std::string &, const std::string &)
{
    auto now = std::time(nullptr);
    out << "<h1>Server time</h1>"
        << "<p>" << std::ctime(&now) << "</p>";
}

// In main():
http::register_html_get_action("time", time_page);
// URL: http://localhost:12345/time
```

Plain-text handler (`register_text_get_action`)

Identical to the HTML handler but the response carries `Content-Type: text/plain`. Ideal for AJAX endpoints that return numbers or short strings rather than full HTML documents.

```
int counter = 0;

void increment(std::ostream & out,
    const std::string &, const std::string &)
```

```
{
    out << ++counter;
}

http::register_text_get_action("increment", increment);
```

Generic handler (`register_generic_get_action`)

Your function is responsible for writing the **complete** HTTP response, including headers, and for flushing the stream. Use this when you need a non-standard MIME type, custom headers, or a long-lived connection (SSE).

```
void download(std::ostream & out,
    const std::string &, const std::string &)
{
    std::string body = "col1,col2\n1,2\n";
    out << http::header("text/csv", body.size())
        << body
        << std::flush;
}

http::register_generic_get_action("data_csv", download);
```

URL parameters

Parameters are passed as the third argument (`params`) to any GET handler. Use `http::decode_params()` to parse them into a key→value map:

```
// URL: http://localhost:12345/calculate?x=10&y=20
void calculate(std::ostream & out,
    const std::string &, const std::string & params)
{
    auto p = http::decode_params(params, false);
    int x = std::stoi(p["x"]);
    int y = std::stoi(p["y"]);
    out << "<p>" << x << " + " << y << " = " << (x + y) << "</p>";
}

http::register_html_get_action("calculate", calculate);
```

The second argument to `decode_params` controls URL-decoding of values: pass `true` when the values may contain percent-encoded characters (form submissions), `false` when the parameters are plain ASCII numbers or identifiers.

POST handlers

POST handlers receive the request body through an `std::istream`.

HTML POST handler (`register_html_post_action`)

Signature:

```
void handler(std::ostream & out,
             const std::string & path,
             const std::string & params,
```

```
            std::istream & in,
            std::size_t content_length,
            const std::string & content_type);
```

Your function **must** read exactly `content_length` bytes from `in`.

Example — processing an HTML form:

```
void greet(std::ostream & out,
    const std::string &, const std::string &,
    std::istream & in, std::size_t len, const std::string &)
{
    std::vector<char> data(len);
    in.read(data.data(), len);

    auto p = http::decode_params(data, true);
    out << "<h1>Hello, "
        << http::html_encode(p["name"])
        << "!</h1>";
}

http::register_html_post_action("greet", greet);
```

Note the use of `http::html_encode()` — always escape user-supplied strings before placing them in HTML output.

Generic POST handler (`register_generic_post_action`) and plain-text POST handler (`register_text_post_action`) follow the same pattern with the same trade-offs as their GET equivalents.

Server-Sent Events (SSE)

Server-Sent Events allow the server to push data to the browser continuously over a single, long-lived HTTP connection. This pattern requires a generic GET handler together with a connection callback.

How it works

1. The browser opens a connection to the SSE endpoint.
2. The generic handler stores a reference to the output stream and returns, keeping the connection open (the thread stays alive).
3. A background thread periodically writes SSE-formatted messages and flushes the stream.
4. The connection callback notifies the application when the browser disconnects, so the stored pointer can be cleared safely.

Full example:

```
#include "http_server.h"
#include <iostream>
#include <mutex>
#include <thread>
#include <chrono>

int value = 0;
std::ostream * updates_stream = nullptr;
std::mutex mtx;
```

```cpp
// Called when the browser opens /get_updates
void get_updates(std::ostream & out,
    const std::string &, const std::string &)
{
    std::lock_guard<std::mutex> lck(mtx);
    updates_stream = &out;  // remember the stream for later pushes
}

// Called whenever a connection is created or is about to close
void on_connection(std::ostream &, http::connection_event ev)
{
    if (ev == http::to_be_closed)
    {
        std::lock_guard<std::mutex> lck(mtx);
        updates_stream = nullptr;
    }
}

// Background thread — pushes updates every second
void activity()
{
    while (true)
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::lock_guard<std::mutex> lck(mtx);
        if (updates_stream != nullptr)
        {
            *updates_stream
                << http::header("text/event-stream")
                << "data: " << ++value << "\r\n\r\n"
                << std::flush;
        }
    }
}

int main()
{
    http::register_generic_get_action("get_updates", get_updates);
    http::register_connection_callback(on_connection);

    std::thread th(activity);
    http::server_start(12345, "files", std::cerr);
}
```

On the browser side, subscribe with the standard `EventSource` API:

```javascript
const es = new EventSource("/get_updates");
es.onmessage = e => document.getElementById("value").textContent = e.data;
```

Connection events

`http::register_connection_callback()` accepts a function with the signature:

```cpp
void callback(std::ostream & stream, http::connection_event event);
```

The `connection_event` enum has two values:

`http::just_connected`
: A new client connection has been established. `stream` is the newly created connection stream.

`http::to_be_closed`
: The connection is about to be torn down. Clear any stored pointer to `stream` here to prevent use-after-free.

Only one callback can be active at a time. Registering a new one replaces the previous registration.

Utility Functions

html_encode

```
std::string http::html_encode(const std::string & s);
```

Replaces `<`, `>`, and `&` with their HTML entity equivalents (`<`, `>`, `&`). Always use this when inserting user-supplied or untrusted strings into HTML output.

url_encode / url_decode

```
std::string http::url_encode(const std::string & s);
std::string http::url_decode(const std::string & s);
```

`url_encode` converts a string to its percent-encoded form suitable for use in a URL. Alphanumeric characters and `-`, `_`, `.`, `~` are left unchanged; spaces become `+`; everything else becomes `%hh`. `url_decode` reverses the transformation.

decode_params

```
http::params_map_type
http::decode_params(const std::string & params, bool decode);

http::params_map_type
http::decode_params(const std::vector<char> & params, bool decode);
```

Parses a `key=value&key=value&...` string into an `unordered_map<string,string>`. The `decode` flag controls whether values are URL-decoded. Pass `true` for form POST bodies, `false` for URL query strings whose values are plain identifiers.

header

```
std::string http::header(const std::string & mime_type,
                         std::size_t content_length = 0,
                         bool cache = false);
```

Generates a minimal `HTTP/1.1 200 OK` response header with the specified MIME type and content length. Used inside generic handlers before writing the response body. When `content_length` is 0 (the default) no `Content-Length` field is emitted, which is the correct behaviour for streaming responses such as SSE.

register_safe_directory

```
void http::register_safe_directory(const std::string & dir_prefix);
```

Registers a directory prefix that is considered safe for client file requests. This limits the file paths that the static file server will serve, preventing path-traversal access to directories outside the intended scope.

Quick-Start Examples

The `examples/` directory contains six self-contained programs that demonstrate the main use cases. Each can be built with `make` and accessed at `http://localhost:12345` (or `https://` for the SSL example).

Example 1 — Static file server

The simplest possible usage: serve the contents of a local directory.

```
#include "http_server.h"

int main()
{
    http::server_start(12345, "files");
}
```

Example 1-SSL — HTTPS static file server

Identical to Example 1, with TLS added.

```
#include "http_server.h"

int main()
{
    http::server_start(12345, "files", {"server.crt", "server.key"});
}
```

Example 2 — Dynamic GET handler

Mix one dynamic page with static file fallback.

```
#include "http_server.h"
#include <iostream>
#include <cstdlib>

void my_action(std::ostream & out,
    const std::string &, const std::string &)
{
    out << "<h1>Random: " << std::rand() << "</h1>";
}

int main()
{
    http::register_html_get_action("my_action", my_action);
    http::server_start(12345, "files", std::cerr);
}
```

Example 3 — URL parameters

Read values from the query string (`/calculate?x=3&y=4`).

```
void calculate(std::ostream & out,
    const std::string &, const std::string & params)
{
    auto p = http::decode_params(params, false);
    int x = std::stoi(p["x"]), y = std::stoi(p["y"]);
    out << "<p>" << x << " + " << y << " = " << (x + y) << "</p>";
}

http::register_html_get_action("calculate", calculate);
```

Example 4 — AJAX

Text endpoints return small values; the HTML page updates them via JavaScript without a full page reload.

```
int value = 0;

void up(std::ostream & out, const std::string &, const std::string &)
{ out << ++value; }

void down(std::ostream & out, const std::string &, const std::string &)
{ out << --value; }

http::register_text_get_action("up",   up);
http::register_text_get_action("down", down);
```

Example 5 — HTML form POST

Handle a form submission; always HTML-encode user input.

```
void greet(std::ostream & out,
    const std::string &, const std::string &,
    std::istream & in, std::size_t len, const std::string &)
{
    std::vector<char> data(len);
    in.read(data.data(), len);
    auto p = http::decode_params(data, true);
    out << "<h1>Hello, " << http::html_encode(p["name"]) << "!</h1>";
}

http::register_html_post_action("greet", greet);
```

Example 6 — Server-Sent Events

A background thread pushes a counter to the browser every second. See the full listing in the server-sent-events section above.

API Reference

Server startup

```
void http::server_start(int port, const std::string & base_directory);

void http::server_start(int port, const std::string & base_directory,
    std::ostream & log, unsigned int mask = log_everything);

void http::server_start(int port, const std::string & base_directory,
    const ssl_server_config & ssl,
    std::ostream & log, unsigned int mask = log_everything);
```

Blocks indefinitely. Register all handlers before calling this function.

Handler registration

```
// GET — body only (headers auto-generated)
void http::register_html_get_action (const char * name, get_action_type f);
void http::register_text_get_action (const char * name, get_action_type f);

// GET — full response (headers + body + flush)
void http::register_generic_get_action(const char * name, get_action_type f);

// POST — body only
void http::register_html_post_action (const char * name, post_action_type f);
void http::register_text_post_action (const char * name, post_action_type f);

// POST — full response
void http::register_generic_post_action(const char * name, post_action_type f);

// Connection lifecycle notifications
void http::register_connection_callback(connection_callback_type f);

// Restrict static file serving to a given directory prefix
void http::register_safe_directory(const std::string & dir_prefix);
```

Callback signatures

```
// GET handler
using get_action_type =
    std::function<void(std::ostream & out,
                       const std::string & path,
                       const std::string & params)>;

// POST handler
using post_action_type =
    std::function<void(std::ostream & out,
                       const std::string & path,
                       const std::string & params,
                       std::istream & in,
                       std::size_t content_length,
                       const std::string & content_type)>;

// Connection callback
using connection_callback_type =
    std::function<void(std::ostream & stream,
                       connection_event event)>;
```

Utilities

```
std::string http::html_encode(const std::string & s);
std::string http::url_encode (const std::string & s);
std::string http::url_decode (const std::string & s);

http::params_map_type
http::decode_params(const std::string &    params, bool decode);
http::params_map_type
http::decode_params(const std::vector<char> & params, bool decode);

std::string http::header(const std::string & mime_type,
                         std::size_t content_length = 0,
                         bool cache = false);
```

SSL configuration (HTTPS only)

```
struct http::ssl_server_config
{
    std::string cert_file;  // path to PEM certificate
    std::string key_file;   // path to PEM private key
};
```