https://simplifycpp.org

DRAFT

Modern C++ Backend Engineering with Boost.Asio on Windows

Prepared by Ayman Alheraki

Modern C++ Backend Engineering with Boost.Asio on Windows

Some drafting assistance and idea exploration were supported by modern AI tools, with full supervision, verification, correction, and authorship.

Prepared by Ayman Alheraki

March 2026

Contents

Authors Introduction

This booklet was written for developers who already understand C++ well and do not need another theoretical explanation of networking concepts. Instead, it focuses on something far more practical: building real backend systems using Boost.Asio on Windows, with working code as the primary teaching method.

Boost.Asio is often introduced as a networking library, but in practice it is much more than that. It provides a complete asynchronous execution model built around `io_context`, where operations are initiated and completed later through handlers or coroutines. This model is powerful enough to serve as the foundation for scalable backend systems.

Rather than explaining this model in abstract terms, this booklet demonstrates it directly through code. Each concept is introduced through a working example, extended step by step, and then refined into a reusable pattern. The goal is that every chapter leaves the reader with something concrete that can be reused in real projects.

The structure of the booklet reflects this philosophy. It starts with a minimal asynchronous example, quickly moves to building a TCP server, and then evolves into HTTP services using Boost.Beast. From there, it introduces routing, middleware, and coroutine-based design, culminating in a small Express-like framework implemented entirely in Modern C++.

This is not a beginners guide. It assumes familiarity with modern C++ features such as smart pointers, lambdas, and RAII. It also assumes that Boost is already installed and usable. The focus is entirely on how to use these tools effectively to build backend systems.

If there is a single goal behind this booklet, it is this:

To show that Modern C++ can be used to build backend systems that are not only fast and efficient, but also clean, structured, and enjoyable to write.

Ayman alheraki

Preface

Most backend development today is associated with high-level ecosystems such as Node.js, Python, or Go. These environments provide simple abstractions for routing, middleware, and request handling, making them easy to use and widely adopted.
C++ is rarely seen in this space, not because it lacks capability, but because it lacks simple, example-driven guidance.
Boost.Asio changes that.
At its core, Boost.Asio provides a consistent asynchronous model where operations are executed independently and completed through well-defined mechanisms. This makes it possible to build highly scalable servers without relying on thread-per-connection designs.
When combined with Boost.Beast, which provides HTTP and WebSocket support, it becomes possible to build complete backend systems in Modern C++.
This booklet does not attempt to compete with full-scale frameworks. Instead, it focuses on showing how such frameworks can be built using simple, understandable components.
The emphasis is always on code:

- minimal explanation
- clear examples
- incremental improvements
- practical patterns

By the end of this booklet, the reader will have built:

- a TCP server
- an HTTP server
- a routing system
- middleware pipeline
- coroutine-based backend
- a small Express-like framework

All using Modern C++ on Windows.

Part I

Core Model (Minimal Theory)

The Asio Execution Model in One Example

1.1 Why Begin with a Timer

The cleanest way to understand the Boost.Asio execution model is to begin with a timer rather than a socket. A timer removes protocol details, buffering concerns, endpoint resolution, and transport-specific error handling, leaving only the essential asynchronous machinery visible. In one short example, the programmer can observe the four fundamental elements that recur throughout backend development with Boost.Asio:

1. an execution context that owns the event-processing machinery,
2. an I/O object associated with that context,
3. an asynchronous initiating call that starts work and returns immediately,
4. a call to `run()` that drives completion.

This is not merely a beginner example. It is the same execution model that later powers timers, sockets, acceptors, HTTP servers, SSL streams, coroutines, composed operations, and user-defined asynchronous abstractions. If this first example is understood deeply, the remainder of the library becomes much easier to reason about.

1.2 Single-file async timer

1.2.1 Complete example

The following program is the smallest realistic asynchronous Boost.Asio program worth studying on Windows. It uses `boost::asio::steady_timer`, which is the correct modern timer type for measuring relative durations because it is based on a monotonic clock and is therefore not affected by wall-clock adjustments.

```
#include <iostream>
#include <chrono>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using boost::system::error_code;

static void on_timer(const error_code& ec)
{
    if (ec)
    {
        std::cout << "timer error: " << ec.message() << '\n';
        return;
    }

    std::cout << "timer expired\n";
}

int main()
{
    try
```

```
    {
        asio::io_context io;

        asio::steady_timer timer(io);
        timer.expires_after(std::chrono::seconds(2));

        std::cout << "starting async wait\n";

        timer.async_wait(&on_timer);

        std::cout << "running io_context\n";
        io.run();

        std::cout << "io_context finished\n";
    }
    catch (const std::exception& ex)
    {
        std::cerr << "fatal error: " << ex.what() << '\n';
        return 1;
    }

    return 0;
}
```

1.2.2 What happens in this program

The execution order is simple but extremely important:

1. `asio::io_context io;` constructs the event-processing context.
2. `asio::steady_timer timer(io);` binds the timer to that context.

3. `timer.expires_after(...);` sets the expiration time.
4. `timer.async_wait(&on_timer);` starts an asynchronous operation and returns immediately.
5. `io.run();` enters the event loop and blocks the current thread until the outstanding work completes.
6. when the timer expires, Asio schedules the completion handler,
7. the handler runs on a thread currently inside `io.run()`,
8. once the handler returns and no more work remains, `io.run()` returns.

This is the full core model in compact form. Nearly every nontrivial Boost.Asio server is an extension of this flow.

1.2.3 Why the example matters

This program proves several essential rules:

- calling an `async_` function does not itself execute the handler,
- asynchronous work does not complete by magic; the context must be driven,
- completion handlers are tied to the execution context,
- the thread that calls `run()` is the thread that executes ready handlers unless multiple threads are running the same context.

1.2.4 Windows compile guidance

For this example, Boost.Asio is used in its normal header-based form. A timer-only example usually does not require separately built Boost libraries.
If Boost is unpacked at:

```
C:\Libraries\boost_1_89_0
```

then an MSVC Developer Command Prompt build may look like this:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 timer_example.cpp
```

A MinGW-w64 build may look like this:

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 timer_example.cpp -o
↪  timer_example.exe
```

If a project later uses libraries that require separate linking, Boost's Windows setup documentation recommends building binaries with Boost.Build from a Developer Command Prompt using commands such as the following from the Boost root directory:

```
bootstrap.bat
b2
```

1.2.5 Expected output

A normal run will produce output similar to the following:

```
starting async wait
running io_context
timer expired
io_context finished
```

The important observation is that `timer expired` appears only after `io.run()` has begun driving the context.

1.3 `io_context` lifecycle

1.3.1 Role of `io_context`

The `io_context` object is the execution engine at the heart of classic Boost.Asio programs. All asynchronous objects are associated with some execution context, and asynchronous completion is delivered only while the context is being actively run.
Conceptually, `io_context` moves through a lifecycle:

1. construction,
2. association of I/O objects and submission of work,
3. execution through `run()` or related functions,
4. exhaustion of outstanding work,
5. stopped state,
6. optional restart and reuse.

1.3.2 Construction

Construction is cheap and explicit:

```
boost::asio::io_context io;
```

At this moment there are no asynchronous operations in flight. Therefore, if `io.run()` is called immediately, it returns immediately because there is no work to process.

1.3.3 A context with no work returns immediately

This behavior often surprises newcomers:

```cpp
#include <iostream>
#include <boost/asio.hpp>

int main()
{
    boost::asio::io_context io;

    std::cout << "before run\n";
    io.run();
    std::cout << "after run\n";
}
```

Output:

```
before run
after run
```

This immediate return is correct. The context had nothing to do.

1.3.4 Outstanding work keeps `run()` alive

Once an asynchronous operation is started, the context has work. In the timer example, the outstanding wait operation prevents `run()` from returning until the timer completes and its handler finishes.
This explains an important practical rule:

> Always ensure that useful work has been submitted before expecting `run()` to remain active.

1.3.5 Completion drains work

When the last outstanding operation completes and its handler returns, `run()` exits. This is one of the cleanest properties of the model: once there is no more work, the event loop naturally drains and returns to the caller.
This is why small single-threaded utilities can be written very naturally with Asio. The call to `run()` acts as the whole execution loop.

1.3.6 Stopped state and reuse

After `run()` exits normally because the context ran out of work, the context is considered stopped. If the programmer wants to submit new work and call `run()` again, the context must first be restarted.

```
#include <iostream>
#include <chrono>
#include <boost/asio.hpp>

namespace asio = boost::asio;

int main()
{
    asio::io_context io;

    asio::steady_timer t1(io);
    t1.expires_after(std::chrono::milliseconds(300));
    t1.async_wait([](const boost::system::error_code& ec)
    {
        if (!ec)
            std::cout << "first wait\n";
```

```
    });

    io.run();

    io.restart();

    asio::steady_timer t2(io);
    t2.expires_after(std::chrono::milliseconds(300));
    t2.async_wait([](const boost::system::error_code& ec)
    {
        if (!ec)
            std::cout << "second wait\n";
    });

    io.run();
}
```

Without `io.restart();`, the second `run()` would not behave as intended because the context remains in the stopped state after the first run completes.

1.3.7 Stopping explicitly

The member function `stop()` requests that the event loop stop as soon as possible. This is useful for shutdown paths, but it must be understood precisely.
Calling `stop()` is not the same as letting all work finish naturally. It causes the event-processing loop to return promptly. Therefore it is appropriate for cancellation-oriented shutdown, not for graceful draining when the program still wants already-ready handlers to continue normally.

```
#include <iostream>
#include <thread>
```

```cpp
#include <chrono>
#include <boost/asio.hpp>

namespace asio = boost::asio;

int main()
{
    asio::io_context io;

    asio::steady_timer timer(io);
    timer.expires_after(std::chrono::seconds(5));
    timer.async_wait([](const boost::system::error_code& ec)
    {
        std::cout << "handler ec = " << ec.message() << '\n';
    });

    std::thread runner([&io]()
    {
        io.run();
    });

    std::this_thread::sleep_for(std::chrono::seconds(1));
    io.stop();

    runner.join();
}
```

In real servers, `stop()` is typically paired with explicit cancellation, socket closure, or destruction of pending I/O objects during shutdown.

1.3.8 Keeping a background context alive

A common backend pattern is to launch an `io_context` on a thread before actual work is submitted. In that case, a work guard is needed so that the event loop does not return immediately.

```
#include <iostream>
#include <thread>
#include <chrono>
#include <boost/asio.hpp>

namespace asio = boost::asio;

int main()
{
    asio::io_context io;

    auto guard = asio::make_work_guard(io);

    std::thread worker([&io]()
    {
        io.run();
    });

    asio::post(io, []()
    {
        std::cout << "posted task executed\n";
    });

    std::this_thread::sleep_for(std::chrono::milliseconds(200));
```

```

    guard.reset();

    worker.join();
}
```

This pattern is extremely common in production systems. The guard keeps the context alive even when there is a temporary gap between operations. Once the guard is reset and all remaining work finishes, `run()` is allowed to exit naturally.

1.3.9 Exceptions from handlers

If a handler throws, the exception propagates through the thread currently executing `run()` or one of its related functions. Therefore, robust backend code should ensure that exceptions are either handled inside handlers or caught around the `run()` loop.

```
#include <iostream>
#include <stdexcept>
#include <boost/asio.hpp>

namespace asio = boost::asio;

int main()
{
    asio::io_context io;

    asio::post(io, []()
    {
        throw std::runtime_error("handler failure");
    });
```

```

    for (;;)
    {
        try
        {
            io.run();
            break;
        }
        catch (const std::exception& ex)
        {
            std::cerr << "caught: " << ex.what() << '\n';
        }
    }
}
```

For backend services on Windows, this outer guard around `run()` is a useful last line of defense.

1.4 Handler execution flow

1.4.1 The three-event model

The documented asynchronous model can be understood as three major events with two phases between them:

1. the initiating function starts the asynchronous operation,
2. the operation becomes outstanding,
3. externally observable side effects become fully established and the completion handler is submitted to an executor,

4. the operation is now complete,
5. the completion handler is invoked with the result.

This structure is fundamental. It explains why starting an asynchronous operation and executing its handler are separate moments.

1.4.2 Initiating function

Every function whose name begins with `async_` is an initiating function. In the timer example, this is:

```
timer.async_wait(&on_timer);
```

This call does not block waiting for expiry. It merely initiates the operation and returns immediately.

1.4.3 Outstanding phase

After initiation, the operation is outstanding. During this phase, the timer is waiting for one of two conditions:

1. the expiry time is reached,
2. the timer is cancelled.

No completion handler is invoked yet. The operation simply exists as work known to the context.

1.4.4 Completion and submission

When the timer expires, the wait operation completes. At this point, the associated completion handler is submitted for execution through the Asio execution machinery.
One subtle but important guarantee for `async_wait()` is that the handler is not invoked from within the initiating function itself, even if completion would be immediate. This prevents accidental inline re-entrancy at the point of initiation and makes control flow much more predictable.

1.4.5 Handler invocation

The handler is then called by a thread that is actively executing one of the event-processing functions such as:

```
run()
run_one()
run_for()
run_until()
poll()
poll_one()
```

In the single-threaded timer example, the same main thread that called `io.run()` invokes the handler. In a multithreaded design, if several threads call `run()` on the same context, any one of those threads may execute a ready handler unless additional serialization mechanisms such as strands are used.

1.4.6 Exactly-once completion for a timer wait

For each call to `async_wait()`, the associated completion handler is called exactly once. There are two primary outcomes:

1. success, where the timer expires normally,
2. cancellation, where the handler receives `boost::asio::error::operation_aborted`.

A strong backend habit is therefore to check the error code first:

```
timer.async_wait([](const boost::system::error_code& ec)
{
    if (ec == boost::asio::error::operation_aborted)
    {
        std::cout << "timer was cancelled\n";
        return;
    }

    if (ec)
    {
        std::cout << "other error: " << ec.message() << '\n';
        return;
    }

    std::cout << "timer fired normally\n";
});
```

1.4.7 Cancellation example

The following example shows the cancellation path clearly:

```
#include <iostream>
#include <thread>
#include <chrono>
#include <boost/asio.hpp>
```

```
5
6  namespace asio = boost::asio;
7  using boost::system::error_code;
8
9  int main()
10 {
11     asio::io_context io;
12     asio::steady_timer timer(io);
13
14     timer.expires_after(std::chrono::seconds(5));
15
16     timer.async_wait([](const error_code& ec)
17     {
18         if (ec == asio::error::operation_aborted)
19             std::cout << "wait cancelled\n";
20         else if (!ec)
21             std::cout << "wait completed\n";
22         else
23             std::cout << "error: " << ec.message() << '\n';
24     });
25
26     std::thread canceller([&timer]()
27     {
28         std::this_thread::sleep_for(std::chrono::seconds(1));
29         timer.cancel();
30     });
31
32     io.run();
33     canceller.join();
```

```
}
```

Expected output:

```
wait cancelled
```

1.4.8 Why handlers run where they do

This design gives Asio a consistent execution model:

- work is initiated by the calling code,
- completion is delivered by the execution context,
- handler execution is centralized around `run()` and its related functions,
- applications can scale from one thread to many threads without changing the meaning of initiation.

That uniformity is one of the main reasons Boost.Asio scales so well from tiny examples to industrial servers.

1.4.9 A precise mental model

For practical backend engineering, the following mental model is reliable:

> An `async_` call starts work and returns immediately. The work remains owned by Asio until it completes. Completion does not directly call the handler from the initiating line. Instead, the completion is queued into the execution system, and the handler runs only when some thread is actively pumping the associated `io_context`.

Once this model becomes intuitive, later topics such as socket accept loops, composed operations, deadlines, backpressure handling, strand-based serialization, and coroutine wrappers become straightforward extensions rather than separate concepts.

1.5 Single-file demo with annotated flow

The next listing combines the essential ideas into one Windows-friendly file with comments focused on execution order.

```cpp
#include <iostream>
#include <chrono>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using boost::system::error_code;

int main()
{
    asio::io_context io;

    // 1. Construct timer and associate it with the io_context.
    asio::steady_timer timer(io);

    // 2. Set relative expiry.
    timer.expires_after(std::chrono::seconds(2));

    // 3. Initiate asynchronous operation.
    //    This returns immediately. No handler runs yet.
    timer.async_wait([](const error_code& ec)
    {
        // 5. This code runs only while some thread is executing io.run().
        if (ec)
        {
            std::cout << "completion with error: " << ec.message() << '\n';
```

```
            return;
        }

        std::cout << "completion: timer expired\n";
    });

    std::cout << "before io.run()\n";

    // 4. Drive the execution context.
    //    The current thread blocks here until all work is done.
    io.run();

    // 6. Once the handler has returned and no work remains, run() exits.
    std::cout << "after io.run()\n";
}
```

1.6 Common beginner mistakes on Windows

1.6.1 Forgetting to call `run()`

The most common mistake is initiating an asynchronous operation and then allowing the program to exit without ever calling `run()`.

```
boost::asio::io_context io;
boost::asio::steady_timer timer(io);
timer.expires_after(std::chrono::seconds(1));
timer.async_wait([](const boost::system::error_code&) {
    std::cout << "never printed\n";
});
```

The handler never runs because the context is never driven.

1.6.2 Letting the timer object die too early

Asynchronous objects must remain alive long enough for their operations to complete. A timer destroyed before completion cancels its outstanding waits.

```
#include <boost/asio.hpp>

void bad_example(boost::asio::io_context& io)
{
    boost::asio::steady_timer timer(io);
    timer.expires_after(std::chrono::seconds(1));
    timer.async_wait([](const boost::system::error_code&) {});
} // timer dies here
```

In larger systems, lifetime management is one of the most important parts of correct Asio design.

1.6.3 Expecting asynchronous code to behave like synchronous code

Code after an `async_` call continues immediately. This means the following assumption is incorrect:

```
timer.async_wait(handler);
std::cout << "this line runs after the timer\n";
```

In reality, that line runs immediately after initiation, not after completion.

1.6.4 Reusing a stopped context without restart

Once a context has stopped because work is exhausted, call `restart()` before running it again for a new batch of work.

1.7 Practical backend significance

The timer example may appear tiny, but it already teaches the exact mechanics required for backend systems on Windows:

- an acceptor waits asynchronously for new clients,
- a socket waits asynchronously for incoming bytes,
- a write operation waits asynchronously for transmission completion,
- a deadline timer waits asynchronously for timeout enforcement,
- all completions return through the same execution machinery.

Thus, replacing the timer with a socket does not change the execution model. It only changes the kind of asynchronous operation being initiated.
This is why the timer example should be treated as foundational rather than trivial.

1.8 Key takeaways

- `io_context` is the engine that drives asynchronous completion.
- An `async_` call starts work and returns immediately.
- Handlers run only on threads actively pumping the context.
- `run()` returns when no work remains.
- Reusing a drained context requires `restart()`.

Your First TCP Server (Step-by-Step)

2.1 Introduction

After understanding the timer-based execution model, the next logical step is to move to TCP. A TCP server introduces three new realities that do not appear in the simple timer example:

1. a listening socket must wait for incoming connections,
2. each accepted client connection has its own socket state,
3. useful work usually continues after accept, through repeated asynchronous reads and writes.

Despite these additions, the underlying model does not change. The same `io_context` drives everything. The same initiating functions start asynchronous operations and return immediately. The same completion handlers are invoked only while some thread is executing `io_context::run()` or a related event-processing function. This continuity is the key reason Boost.Asio scales from tiny examples to production backend services.
This chapter develops a TCP server in four stages:

1. a minimal blocking single-client server to establish the socket roles,
2. an asynchronous acceptor that no longer blocks the main flow,

3. an asynchronous read/write session loop for a connected client,
4. a final multi-client server where each connection is handled independently.

All examples are designed for Windows development with modern Boost.Asio and standard C++20-style compilation.

2.2 Minimal server (single client)

2.2.1 Why start with a synchronous baseline

Even though the goal of this book is asynchronous backend engineering, a tiny synchronous baseline is useful. It makes the purpose of the two TCP objects immediately visible:

- `tcp::acceptor` listens on a local endpoint,
- `tcp::socket` represents one connected client.

The synchronous form also shows clearly that `accept()` blocks until a client arrives, and that `read_some()` blocks until data becomes available.
This model is simple, but it does not scale well. A single thread becomes trapped waiting on one client at a time. That limitation is exactly why asynchronous design matters.

2.2.2 Complete minimal synchronous server

The following example accepts one client, reads one message, sends one reply, and exits.

```
#include <array>
#include <iostream>
#include <string>
#include <boost/asio.hpp>
```

```
5
6  namespace asio = boost::asio;
7  using asio::ip::tcp;
8
9  int main()
10 {
11     try
12     {
13         asio::io_context io;
14
15         tcp::acceptor acceptor(io, tcp::endpoint(tcp::v4(), 5555));
16
17         std::cout << "Server listening on port 5555...\n";
18
19         tcp::socket socket(io);
20         acceptor.accept(socket);
21
22         std::cout << "Client connected from "
23                   << socket.remote_endpoint() << '\n';
24
25         std::array<char, 1024> data{};
26         std::size_t n = socket.read_some(asio::buffer(data));
27
28         std::cout << "Received: "
29                   << std::string(data.data(), n) << '\n';
30
31         std::string reply = "Hello from synchronous server\n";
32         asio::write(socket, asio::buffer(reply));
33
```

```
        std::cout << "Reply sent. Server exiting.\n";
    }
    catch (const std::exception& ex)
    {
        std::cerr << "Fatal error: " << ex.what() << '\n';
        return 1;
    }

    return 0;
}
```

2.2.3 What this program teaches

This first server teaches four foundational facts:

1. The `acceptor` owns the listening port.
2. The `socket` for a client is distinct from the acceptor.
3. Once the client is accepted, all further communication is done on the connected socket.
4. The model is easy to understand, but every operation blocks the thread.

From a backend engineering perspective, the final point is the most important. A real service cannot afford to wait on one connection while ignoring the rest of the world.

2.2.4 Windows build guidance

Assuming Boost is unpacked at:

```
C:\Libraries\boost_1_89_0
```

an MSVC Developer Command Prompt build may look like this:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 sync_server.cpp
```

A MinGW-w64 build may look like this:

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 sync_server.cpp -o
↪  sync_server.exe
```

2.2.5 A quick test from Windows

Open one terminal to run the server. Then from another terminal use a TCP client such as:

```
telnet 127.0.0.1 5555
```

or, if a command-line netcat variant is installed:

```
nc 127.0.0.1 5555
```

Type a line and press Enter. The server reads the incoming data, prints it, sends a reply, and exits.

2.3 Add async accept

2.3.1 Why asynchronous accept matters

The first major transformation is to replace blocking `accept()` with `async_accept()`. This changes the server from a thread-blocking design into an event-driven design.
Conceptually, the server now does this:

1. create an acceptor bound to a listening endpoint,

2. start one asynchronous accept operation,

3. return immediately to the event loop,

4. let `io_context` invoke a completion handler when a client arrives.

The core execution rule from Chapter 1 is unchanged: the accept completion handler is executed only while some thread is pumping the context.

2.3.2 Single asynchronous accept example

The following example accepts one client asynchronously and then sends a small greeting.

```
#include <iostream>
#include <memory>
#include <string>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class one_accept_server
{
public:
    explicit one_accept_server(asio::io_context& io)
        : acceptor_(io, tcp::endpoint(tcp::v4(), 5555))
    {
    }

    void start()
```

```
    {
        std::cout << "Async server listening on port 5555...\n";

        acceptor_.async_accept(
            [this](error_code ec, tcp::socket socket)
            {
                if (ec)
                {
                    std::cerr << "accept error: " << ec.message() << '\n';
                    return;
                }

                std::cout << "Accepted client from "
                          << socket.remote_endpoint() << '\n';

                auto message =
                 std::make_shared<std::string>("Hello from async accept server\n");

                asio::async_write(
                    socket,
                    asio::buffer(*message),
                  [message, s = std::move(socket)](error_code write_ec, std::size_t)
                    {
                        if (write_ec)
                        {
                            std::cerr << "write error: "
                                      << write_ec.message() << '\n';
                            return;
                        }
```

```
48
49                          std::cout << "Greeting sent\n";
50                      });
51              });
52      }
53
54  private:
55      tcp::acceptor acceptor_;
56  };
57
58  int main()
59  {
60      try
61      {
62          asio::io_context io;
63
64          one_accept_server server(io);
65          server.start();
66
67          io.run();
68      }
69      catch (const std::exception& ex)
70      {
71          std::cerr << "Fatal error: " << ex.what() << '\n';
72          return 1;
73      }
74
75      return 0;
76  }
```

2.3.3 What changed

Several important changes have occurred:

1. The server no longer blocks at the point of calling `async_accept()`.
2. The accepted socket is delivered to the handler when a client arrives.
3. The server logic is now split into phases connected by completion handlers.
4. The socket must remain alive until the asynchronous write finishes, which is why it is moved into the write handler.

This last point is crucial. Lifetime management becomes one of the central design concerns in Boost.Asio programs.

2.3.4 Why the socket is moved into the handler

A common beginner mistake is to let the accepted socket go out of scope before the asynchronous write completes. Since the socket is a local object of the accept completion handler, it must be preserved beyond the line that starts the write operation. Moving it into the lambda capture is a correct and modern solution.

2.3.5 The role of `io.run()`

Without:

```
io.run();
```

nothing happens. The accept operation is outstanding, but it has no thread available to process completion. The event loop is therefore not optional; it is the execution engine that turns outstanding work into handler invocation.

2.4 Add read/write loop

2.4.1 From one-shot reply to a session

A useful TCP server usually performs repeated operations after accept. Once a client is connected, the server must continue reading requests and writing responses until the session ends.
This requires a per-connection object. Each client connection needs its own socket, buffer, and continuation logic. The standard and most robust pattern is:

1. accept a new client socket,
2. create a session object,
3. start an asynchronous read,
4. when data arrives, process it and start an asynchronous write,
5. after the write completes, start reading again.

This read/write cycle is the real beginning of backend protocol design.

2.4.2 Echo session with asynchronous read/write loop

The following example implements a single-client echo server session. The server accepts one client, then repeatedly reads data and writes it back.

```
#include <array>
#include <iostream>
#include <memory>
#include <boost/asio.hpp>

```

```
namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        do_read();
    }

private:
    void do_read()
    {
        auto self = shared_from_this();

        socket_.async_read_some(
            asio::buffer(data_),
            [self](error_code ec, std::size_t length)
            {
                if (ec)
                {
                    if (ec != asio::error::eof)
```

```
                {
                    std::cerr << "read error: " << ec.message() << '\n';
                }
                return;
            }

            self->do_write(length);
        });
    }

    void do_write(std::size_t length)
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(data_.data(), length),
            [self](error_code ec, std::size_t)
            {
                if (ec)
                {
                    std::cerr << "write error: " << ec.message() << '\n';
                    return;
                }

                self->do_read();
            });
    }

```

```
64 private:
65     tcp::socket socket_;
66     std::array<char, 1024> data_{};
67 };
68 
69 class single_client_server
70 {
71 public:
72     explicit single_client_server(asio::io_context& io)
73         : acceptor_(io, tcp::endpoint(tcp::v4(), 5555))
74     {
75     }
76 
77     void start()
78     {
79         acceptor_.async_accept(
80             [this](error_code ec, tcp::socket socket)
81             {
82                 if (ec)
83                 {
84                     std::cerr << "accept error: " << ec.message() << '\n';
85                     return;
86                 }
87 
88                 std::cout << "Client connected: "
89                           << socket.remote_endpoint() << '\n';
90 
91                 std::make_shared<session>(std::move(socket))->start();
92             });
```

```
    }

private:
    tcp::acceptor acceptor_;
};

int main()
{
    try
    {
        asio::io_context io;

        single_client_server server(io);
        server.start();

        io.run();
    }
    catch (const std::exception& ex)
    {
        std::cerr << "Fatal error: " << ex.what() << '\n';
        return 1;
    }

    return 0;
}
```

2.4.3 Why `enable_shared_from_this` is used

A session object must survive until all of its outstanding asynchronous operations complete. If a handler captured a raw `this` pointer and the session object were destroyed early, the handler would later invoke member functions through a dangling pointer.
Using `std::enable_shared_from_this` solves this in a standard and very common Asio style:

1. the session is created as a `std::shared_ptr<session>`,
2. each asynchronous step captures `self = shared_from_this()`,
3. the object remains alive until the captured shared pointer is released after handler completion.

This is one of the most important object-lifetime patterns in callback-based Boost.Asio server design.

2.4.4 Why `async_read_some` is used here

The member function `async_read_some()` starts a single low-level read operation. It reads whatever bytes are currently available up to the size of the buffer. That makes it ideal for a first TCP server because it exposes the raw nature of stream-oriented networking:

- TCP does not preserve application message boundaries,
- one read may return less than a full logical message,
- one logical message may arrive across multiple reads,
- protocol framing is the application's responsibility.

For an echo server, raw chunks are enough. For text protocols, structured binary protocols, or length-prefixed messages, a higher-level framing strategy is needed.

2.4.5 Why `async_write` is used instead of `socket.async_send`

The free function `asio::async_write()` is a composed operation. It continues issuing lower-level writes until the entire supplied buffer sequence has been written or an error occurs. This is usually what a beginner expects when sending a response.
By contrast, a single low-level send operation may write only part of the data. For first servers, `async_write()` makes the intent explicit and the behavior safer.

2.4.6 Connection shutdown behavior

In the read handler, if the client closes the connection normally, the server often receives `asio::error::eof`. That is not a catastrophic failure. It simply means the peer ended the stream. This distinction matters in real servers:

- normal session end is not the same as transport failure,
- many errors should end only the current session, not the entire server,
- the acceptor and `io_context` usually continue running.

2.5 Final multi-client version

2.5.1 The missing scalability step

The previous example still accepted only one client. Once that single client was accepted, no new `async_accept()` call was initiated. A real server must continuously accept new connections while existing sessions continue their own asynchronous read/write loops.
The correct model is:

1. begin an accept operation,

2. when a client arrives, create a session and start it,
3. immediately begin another accept operation,
4. allow all sessions to progress independently through the same `io_context`.

This is the central architectural shape of many classic asynchronous TCP servers.

2.5.2 Final multi-client echo server

The following program is a clean, modern, single-threaded multi-client TCP echo server. It is simple enough for study, but already expresses the essential architecture used in many real systems.

```
#include <array>
#include <iostream>
#include <memory>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

```

```
    void start()
    {
        try
        {
            std::cout << "Session started for "
                      << socket_.remote_endpoint() << '\n';
        }
        catch (...)
        {
        }

        do_read();
    }

private:
    void do_read()
    {
        auto self = shared_from_this();

        socket_.async_read_some(
            asio::buffer(data_),
            [self](error_code ec, std::size_t length)
            {
                if (ec)
                {
                    if (ec != asio::error::eof)
                    {
                        std::cerr << "read error: "
                                  << ec.message() << '\n';
```

```
                    }
                    return;
                }

                self->do_write(length);
            });
    }

    void do_write(std::size_t length)
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(data_.data(), length),
            [self](error_code ec, std::size_t)
            {
                if (ec)
                {
                    std::cerr << "write error: "
                              << ec.message() << '\n';
                    return;
                }

                self->do_read();
            });
    }

private:
```

```
    tcp::socket socket_;
    std::array<char, 1024> data_{};
};

class server
{
public:
    server(asio::io_context& io, unsigned short port)
        : acceptor_(io, tcp::endpoint(tcp::v4(), port))
    {
    }

    void start()
    {
        do_accept();
    }

private:
    void do_accept()
    {
        acceptor_.async_accept(
            [this](error_code ec, tcp::socket socket)
            {
                if (!ec)
                {
                    std::make_shared<session>(std::move(socket))->start();
                }
                else
                {
```

```
                    std::cerr << "accept error: "
                              << ec.message() << '\n';
                }

                do_accept();
            });
    }

private:
    tcp::acceptor acceptor_;
};

int main()
{
    try
    {
        asio::io_context io;

        server srv(io, 5555);
        srv.start();

        std::cout << "Multi-client async echo server listening on port 5555...\n";

        io.run();
    }
    catch (const std::exception& ex)
    {
        std::cerr << "Fatal error: " << ex.what() << '\n';
        return 1;
```

```
    }

    return 0;
}
```

2.5.3 Why this server supports multiple clients

The server supports multiple clients not because it creates one thread per client, but because it keeps multiple asynchronous operations outstanding at the same time:

- the acceptor always has another accept pending,
- each active session usually has one read or write pending,
- the `io_context` dispatches handlers as operations complete.

Even with one thread calling `run()`, multiple connections can be interleaved efficiently because the thread is not blocked waiting on one specific socket operation.

2.5.4 Handler flow in the final server

For one client, the flow is:

1. accept completes,
2. session object is created,
3. session starts a read,
4. read completes,
5. session starts a write,

6. write completes,
7. session starts another read.

For the server as a whole, the acceptor does this forever:

1. wait asynchronously for a new client,
2. on completion, launch a session,
3. immediately wait for the next client.

This separation between the listener and per-client sessions is a fundamental structural pattern in asynchronous backend systems.

2.5.5 Why calling `do_accept()` again is essential

A surprisingly common mistake is to accept one client successfully and forget to start the next accept. In that situation, the server appears to work for the first client but silently stops admitting additional connections.
The line:

```
do_accept();
```

at the end of the accept handler is therefore not an optional convenience. It is the mechanism that turns one accepted connection into a continuously listening server.

2.5.6 A more defensive accept pattern

Some production code prefers to issue the next accept even when an accept error occurs, unless the server is intentionally shutting down. That pattern is already present in the example above. This is important because transient failures should usually not terminate the whole listener.

2.5.7 What is still missing before production use

This final chapter example is a true asynchronous multi-client server, but it is still deliberately minimal. Before using such a design in production, backend engineers usually add:

- protocol framing rules,
- input size limits,
- connection timeouts,
- structured logging,
- graceful shutdown,
- backpressure management,
- thread-pool execution where needed,
- strand-based serialization when shared state exists,
- per-session write queues when multiple writes may overlap.

Those additions do not change the basic server shape. They refine it.

2.6 Windows usage notes

2.6.1 Testing with two or more clients

This final server is ideal for a simple manual test. Run the server in one terminal and connect from multiple client terminals. Each client can send text, and the server echoes the bytes back. Because the server is asynchronous, one connected client does not prevent another from connecting. This is the first direct demonstration that the design has moved beyond the one-client blocking model.

2.6.2 Firewall and port considerations

On Windows, the first run may trigger a firewall prompt. If local testing is done on:

```
127.0.0.1:5555
```

the exposure is local only. For LAN testing, the chosen port must be allowed through the firewall as appropriate.

2.6.3 Common compile command

For MSVC:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 multi_client_server.cpp
```

For MinGW-w64:

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 multi_client_server.cpp
↪ -o multi_client_server.exe
```

2.7 Common mistakes in first TCP servers

2.7.1 Destroying the session too early

If a session object is not kept alive across asynchronous operations, handlers will eventually access invalid state. This is why the shared-ownership pattern is so common in handler-based Asio code.

2.7.2 Starting overlapping writes on one socket

A first echo server is usually safe because each read leads to one write and then another read. In richer protocols, however, several parts of the program may try to write simultaneously on the same socket. That requires a write queue and serialized write initiation.

2.7.3 Assuming one read equals one message

TCP is a byte stream, not a message protocol. A logical application message may require multiple reads, or several messages may arrive in one read. Protocol framing must always be designed explicitly.

2.7.4 Forgetting that the acceptor and session roles are different

The acceptor listens for new clients. The connected socket communicates with one client. Confusing these roles leads to poor architecture and incorrect code.

2.7.5 Stopping after the first accept

A server that forgets to re-arm `async_accept()` is not really a server. It is a one-shot connection demo.

2.8 Key architectural lessons from this chapter

This chapter establishes the core architecture that will recur throughout the rest of the book:

1. one `io_context` drives all network I/O,
2. one acceptor listens for incoming clients,
3. each client connection is represented by its own session object,
4. each session advances through a chain of asynchronous operations,
5. multi-client support comes from re-arming accept and preserving independent session state.

Once these five ideas are fully understood, more advanced designs such as line-based protocols, binary framed protocols, timeouts, command routers, HTTP servers, and coroutine-based sessions become natural extensions rather than entirely new concepts.

2.9 Very short takeaways

- `tcp::acceptor` listens; `tcp::socket` talks to one client.
- `async_accept()` turns blocking wait into event-driven acceptance.
- A session object is the natural unit of per-client state.
- Re-arming accept is what makes the server multi-client.
- The final structure is the foundation of most classic Boost.Asio TCP servers.

Part II

Real Networking Patterns

Session-Based Server Design

3.1 Introduction

A serious TCP server does not stop at accepting a socket and performing one immediate operation. Once a client is accepted, the server must preserve all state associated with that client, continue issuing asynchronous operations, and guarantee that the connection object stays alive until every outstanding handler has finished using it.

This requirement leads naturally to the session-based design that appears repeatedly in Boost.Asio examples and in real backend systems. In this design, each connected client is represented by a dedicated class instance, often called `session`, `connection`, or a similar per-client abstraction. The server object remains responsible for listening and accepting, while the session object becomes responsible for all per-client read, write, parsing, buffering, timeout, and shutdown logic.

This separation is one of the most important architectural steps in learning Boost.Asio properly. It does not merely improve code organization. It is also the mechanism by which lifetime correctness becomes manageable.

3.2 The problem that session objects solve

3.2.1 Why a local socket is not enough

A beginner often starts with a model like this:

```
acceptor.async_accept(
    [&](boost::system::error_code ec, tcp::socket socket)
    {
        if (!ec)
        {
            socket.async_read_some(...);
        }
    });
```

At first glance this appears reasonable, but it hides an immediate lifetime problem. The `socket` object is local to the accept handler. As soon as the handler returns, that local object is destroyed unless it has been moved into some longer-lived owner. Any further asynchronous operation that expects the socket to remain alive becomes invalid or cancelled.
A single asynchronous operation is therefore not enough to define a robust connection design. The programmer needs an object that owns the socket and all related buffers for the entire lifetime of the client interaction.

3.2.2 Why the server itself should not hold all per-client state

Another tempting design is to let the main server class own every current socket, buffer, and protocol variable directly. This also becomes difficult quickly:

- each client needs independent state,
- reads and writes complete at different times for different clients,

- parsing state may differ per connection,
- cleanup must occur per connection, not for the whole server,
- code becomes tangled when listener logic and protocol logic are mixed together.

The session class solves this by making one object responsible for one client.

3.3 Session class

3.3.1 Core idea

A session class usually owns at least the following:

1. a connected `tcp::socket`,
2. one or more buffers,
3. any protocol state needed for the current client,
4. member functions that launch the next asynchronous step,
5. handlers that continue the session flow.

The server accepts new sockets and creates session objects. Each session object then drives itself through a chain of asynchronous operations.

3.3.2 Minimal session skeleton

The following is the essential minimal skeleton for a Boost.Asio TCP session:

```
#include <array>
#include <memory>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        do_read();
    }

private:
    void do_read()
    {
        auto self = shared_from_this();

        socket_.async_read_some(
            asio::buffer(data_),
            [self](error_code ec, std::size_t length)
```

```
            {
                if (ec)
                    return;

                self->do_write(length);
            });
    }

    void do_write(std::size_t length)
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(data_.data(), length),
            [self](error_code ec, std::size_t)
            {
                if (ec)
                    return;

                self->do_read();
            });
    }

private:
    tcp::socket socket_;
    std::array<char, 1024> data_{};
};
```

This pattern is simple, but it already expresses the main structure used in many asynchronous

servers:

- the session owns the connected socket,
- the session owns the read buffer,
- `start()` begins the first asynchronous step,
- each completion handler launches the next operation,
- the object stays alive by capturing `shared_from_this()`.

3.3.3 Why the constructor takes a socket by value

Modern Boost.Asio examples often pass the accepted socket directly into the session constructor and then move it into the member variable:

```
explicit session(tcp::socket socket)
    : socket_(std::move(socket))
{
}
```

This is a clean C++11-and-later design. The acceptor produces a connected socket object, and ownership is transferred into the new session object. This avoids the older two-step style in which a connection object was created first and then an accept operation filled a socket member through a separate accessor function.
Both styles are valid, but the move-based constructor is usually clearer for modern code.

3.3.4 Why `start()` is separate from the constructor

A common best practice is to keep constructors simple and not launch asynchronous operations from inside the constructor itself. Instead, the object is fully created first, and then a distinct `start()` member begins the asynchronous flow.

This has several benefits:

- the object is fully formed before handlers can observe it,
- ownership can be established clearly before the first operation starts,
- error handling and setup sequencing become easier to reason about,
- the construction phase remains separate from the execution phase.

For these reasons, `start()` is one of the most recognizable features of Boost.Asio session designs.

3.3.5 Session with explicit logging and error handling

The following version is slightly more realistic for backend work on Windows:

```
#include <array>
#include <iostream>
#include <memory>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
```

```
    {
    }

    void start()
    {
        try
        {
            std::cout << "session started for "
                      << socket_.remote_endpoint() << '\n';
        }
        catch (...)
        {
        }

        do_read();
    }

private:
    void do_read()
    {
        auto self = shared_from_this();

        socket_.async_read_some(
            asio::buffer(data_),
            [self](error_code ec, std::size_t length)
            {
                if (ec)
                {
                    if (ec != asio::error::eof)
```

```
                {
                    std::cerr << "read error: "
                              << ec.message() << '\n';
                }
                return;
            }

            std::cout << "received " << length << " bytes\n";
            self->do_write(length);
        });
    }

    void do_write(std::size_t length)
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(data_.data(), length),
            [self](error_code ec, std::size_t bytes_written)
            {
                if (ec)
                {
                    std::cerr << "write error: "
                              << ec.message() << '\n';
                    return;
                }

                std::cout << "sent " << bytes_written << " bytes\n";
```

```
                self->do_read();
            });
    }

private:
    tcp::socket socket_;
    std::array<char, 4096> data_{};
};
```

3.4 `shared_from_this` pattern

3.4.1 Why this pattern exists

The central difficulty in asynchronous code is that initiating a read or write does not complete the operation immediately. The operation becomes outstanding, and its completion handler may run later. Therefore, every object referenced by that handler must still be alive at the time of handler execution.

If a session member function starts an asynchronous operation and the handler captures only a raw `this` pointer, the program silently assumes that the session object will still exist when the handler runs. In real code, that assumption is often wrong.

The `shared_from_this()` pattern solves this by making the handler own a shared reference to the session object until the handler has completed.

3.4.2 Basic form of the pattern

The essential sequence is:

1. derive from `std::enable_shared_from_this<session>`,
2. create the object as a `std::shared_ptr<session>`,

3. inside any asynchronous member function, obtain `auto self = shared_from_this();`,
4. capture `self` in the handler,
5. use `self->...` inside the handler when continuing the chain.

The key line is:

```
auto self = shared_from_this();
```

That line does not copy the whole object. It produces a new `std::shared_ptr` that shares ownership of the same session instance.

3.4.3 Why capturing `self` is better than capturing raw `this`

Compare the unsafe and safe forms.
Unsafe style:

```
void do_read()
{
    socket_.async_read_some(
        asio::buffer(data_),
        [this](error_code ec, std::size_t length)
        {
            if (!ec)
                do_write(length);
        });
}
```

Safe session ownership style:

```
void do_read()
{
    auto self = shared_from_this();

    socket_.async_read_some(
        asio::buffer(data_),
        [self](error_code ec, std::size_t length)
        {
            if (!ec)
                self->do_write(length);
        });
}
```

The second form guarantees that the session object remains alive until the lambda has finished executing. The first form gives no such guarantee.

3.4.4 Important rule: the object must already be owned by `shared_ptr`

The `shared_from_this()` call is valid only when the object is already owned by a `std::shared_ptr`. Therefore, a session should be created like this:

```
auto s = std::make_shared<session>(std::move(socket));
s->start();
```

A session created on the stack or through a raw `new` without immediate shared ownership is not compatible with safe use of `shared_from_this()`.

3.4.5 Factory-style creation pattern

Some code bases prefer a named factory function to make this rule explicit:

```
class session : public std::enable_shared_from_this<session>
{
public:
    static std::shared_ptr<session> create(tcp::socket socket)
    {
        return std::shared_ptr<session>(
            new session(std::move(socket)));
    }

    void start()
    {
        do_read();
    }

private:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void do_read()
    {
        auto self = shared_from_this();

        socket_.async_read_some(
            asio::buffer(data_),
            [self](error_code ec, std::size_t length)
            {
                if (!ec)
```

```
                    self->do_write(length);
                });
    }

    void do_write(std::size_t length)
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(data_.data(), length),
            [self](error_code ec, std::size_t)
            {
                if (!ec)
                    self->do_read();
            });
    }

private:
    tcp::socket socket_;
    std::array<char, 1024> data_{};
};
```

This style resembles the older official tutorial examples that exposed a creation function returning a shared pointer. In newer code, `std::make_shared` is often simpler, but both express the same ownership principle.

3.4.6 What the pattern does not do

The `shared_from_this()` pattern does not solve every design issue automatically. In particular, it does not:

- serialize concurrent handlers,
- prevent logical protocol bugs,
- create message boundaries on TCP streams,
- manage application-level shutdown policy by itself,
- prevent cycles if other objects also keep shared ownership indefinitely.

Its purpose is narrower and extremely important: preserve object lifetime correctly while asynchronous work is outstanding.

3.5 Lifetime correctness (code-focused)

3.5.1 The main principle

In Boost.Asio, correct asynchronous code is fundamentally a lifetime problem. Whenever an operation is initiated, every object referenced later by the completion handler must remain alive until handler execution is complete.
That statement applies not only to session objects, but also to:

- sockets,
- timers,
- buffers,

- strings used by `async_write()`,
- parsed request objects,
- any state captured by lambdas.

The session class is simply the most visible place where this principle appears.

3.5.2 Incorrect example: local response string dies too early

A classic bug looks like this:

```
void send_message()
{
    std::string msg = "hello client\n";

    asio::async_write(
        socket_,
        asio::buffer(msg),
        [](boost::system::error_code, std::size_t)
        {
        });
}
```

The problem is that `msg` is destroyed when `send_message()` returns, but the asynchronous write may complete later. The buffer passed to the operation refers to memory that no longer exists.

3.5.3 Correct example: response owned by shared pointer

One correct fix is to give the outgoing data shared ownership for the duration of the write:

```
void send_message()
{
    auto self = shared_from_this();
    auto msg = std::make_shared<std::string>("hello client\n");

    asio::async_write(
        socket_,
        asio::buffer(*msg),
        [self, msg](boost::system::error_code ec, std::size_t)
        {
            if (ec)
                return;

            self->do_read();
        });
}
```

Now both the session object and the outgoing message survive until the handler returns.

3.5.4 Alternative correct example: message stored as a member

Another common pattern is to store outgoing data inside the session object itself:

```
class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }
```

```
8
9       void start()
10      {
11          outgoing_ = "hello client\n";
12          do_write();
13      }
14
15  private:
16      void do_write()
17      {
18          auto self = shared_from_this();
19
20          asio::async_write(
21              socket_,
22              asio::buffer(outgoing_),
23              [self](boost::system::error_code ec, std::size_t)
24              {
25                  if (ec)
26                      return;
27
28                  self->do_read();
29              });
30      }
31
32      void do_read()
33      {
34      }
35
36  private:
```

```
    tcp::socket socket_;
    std::string outgoing_;
};
```

Because `outgoing_` is a member of the session, and the session is kept alive by `self`, the buffer remains valid.

3.5.5 Incorrect example: stack-allocated session

The following is structurally wrong:

```
void handle_client(tcp::socket socket)
{
    session s(std::move(socket));
    s.start();
}
```

Once `handle_client()` returns, the stack-allocated session is destroyed, even though its asynchronous operations may still be outstanding.

3.5.6 Correct example: heap-owned session

The correct pattern is:

```
void handle_client(tcp::socket socket)
{
    std::make_shared<session>(std::move(socket))->start();
}
```

This ensures that the session object can outlive the scope that created it.

3.5.7 Incorrect example: reusing a buffer before write completion

Another subtle lifetime bug occurs when a write is started using a member buffer that is then immediately overwritten before the write finishes:

```
void unsafe_send_twice()
{
    output_ = "first message";

    asio::async_write(
        socket_,
        asio::buffer(output_),
        [](boost::system::error_code, std::size_t) {});

    output_ = "second message";
}
```

Even though `output_` remains alive, its contents may be modified before the first asynchronous write has completed. Lifetime alone is not enough; data stability during the operation also matters.

3.5.8 Correct approach: serialize writes with stable storage

A safer approach is to queue outgoing messages and never begin the next write until the current one completes:

```
#include <deque>
#include <string>

class session : public std::enable_shared_from_this<session>
{
```

```
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void deliver(std::string msg)
    {
        bool write_in_progress = !write_queue_.empty();
        write_queue_.push_back(std::move(msg));

        if (!write_in_progress)
            do_write();
    }

private:
    void do_write()
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(write_queue_.front()),
            [self](boost::system::error_code ec, std::size_t)
            {
                if (ec)
                    return;

                self->write_queue_.pop_front();
```

```
                if (!self->write_queue_.empty())
                    self->do_write();
            });
    }

private:
    tcp::socket socket_;
    std::deque<std::string> write_queue_;
};
```

This pattern is not only about lifetime. It also preserves correctness of the bytes being transmitted.

3.5.9 Lifetime of read buffers

Read buffers must also remain valid until the read handler is invoked. A member array is one of the simplest correct designs:

```
std::array<char, 4096> data_{};
```

Using a member buffer is usually easier and safer than using a temporary local buffer inside a function that launches an asynchronous read.

3.5.10 Session shutdown and destruction

When a session encounters EOF or another terminal error, the usual minimal design simply stops issuing new operations. Once the last handler returns and no more shared pointers remain, the session object is destroyed automatically.
This is one of the elegant properties of the session-based design:

- the session begins when a shared object is created,
- the session continues while asynchronous operations keep shared ownership alive,
- the session ends naturally when no further operations remain and no ownership remains.

3.5.11 A compact lifetime checklist

For every new asynchronous operation, ask the following:

1. Will the owning object still exist when the handler runs
2. Will every buffer still exist when the handler runs
3. Will the bytes inside those buffers remain unchanged for the duration of the operation
4. Is any raw pointer captured where shared or value ownership is required
5. Can any outer scope return before the asynchronous step completes

This checklist prevents a large percentage of real-world Asio bugs.

3.6 Server plus session integration

3.6.1 The clean division of responsibilities

A robust first architecture separates responsibilities like this:

- the server owns the acceptor,
- the server repeatedly starts `async_accept()`,
- the session owns one connected socket,

- the session manages all further communication for that client.

This division keeps the listener small and keeps per-client logic isolated.

3.6.2 Complete reusable server plus session pattern

The following listing is a reusable foundation for many small and medium TCP servers on Windows:

```
#include <array>
#include <iostream>
#include <memory>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        do_read();
    }
```

```
22
23  private:
24      void do_read()
25      {
26          auto self = shared_from_this();
27
28          socket_.async_read_some(
29              asio::buffer(data_),
30              [self](error_code ec, std::size_t length)
31              {
32                  if (ec)
33                  {
34                      if (ec != asio::error::eof)
35                      {
36                          std::cerr << "read error: "
37                                    << ec.message() << '\n';
38                      }
39                      return;
40                  }
41
42                  self->do_write(length);
43              });
44      }
45
46      void do_write(std::size_t length)
47      {
48          auto self = shared_from_this();
49
50          asio::async_write(
```

```
            socket_,
            asio::buffer(data_.data(), length),
            [self](error_code ec, std::size_t)
            {
                if (ec)
                {
                    std::cerr << "write error: "
                              << ec.message() << '\n';
                    return;
                }

                self->do_read();
            });
    }

private:
    tcp::socket socket_;
    std::array<char, 4096> data_{};
};

class server
{
public:
    server(asio::io_context& io, unsigned short port)
        : acceptor_(io, tcp::endpoint(tcp::v4(), port))
    {
    }

    void start()
```

```
80      {
81          do_accept();
82      }
83
84  private:
85      void do_accept()
86      {
87          acceptor_.async_accept(
88              [this](error_code ec, tcp::socket socket)
89              {
90                  if (!ec)
91                  {
92                      std::make_shared<session>(
93                          std::move(socket))->start();
94                  }
95                  else
96                  {
97                      std::cerr << "accept error: "
98                                << ec.message() << '\n';
99                  }
100
101                 do_accept();
102             });
103     }
104
105 private:
106     tcp::acceptor acceptor_;
107 };
108
```

```
int main()
{
    try
    {
        asio::io_context io;

        server srv(io, 5555);
        srv.start();

        std::cout << "server listening on port 5555\n";
        io.run();
    }
    catch (const std::exception& ex)
    {
        std::cerr << "fatal error: " << ex.what() << '\n';
        return 1;
    }

    return 0;
}
```

3.6.3 Why this pattern is reusable

This pattern is reusable because the accept side and the session side are already cleanly separated. To evolve it into a real backend server, the programmer can extend the session object with:

- line-based framing,
- binary message framing,

- command dispatch,
- authentication state,
- timeout timers,
- write queues,
- protocol parsers,
- metrics and structured logging.

The outer structure rarely needs to change. In practice, most server growth happens inside the session object and around its coordination with shared services.

3.7 Windows build guidance

3.7.1 Typical MSVC command

Assuming Boost headers are located at:

```
C:\Libraries\boost_1_89_0
```

a typical MSVC Developer Command Prompt build is:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 session_server.cpp
```

3.7.2 Typical MinGW-w64 command

With MinGW-w64 on Windows:

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 session_server.cpp -o
↪  session_server.exe
```

3.7.3 Testing

Run the server in one terminal and connect from another terminal with a TCP client. A simple local test may use:

```
telnet 127.0.0.1 5555
```

or a netcat-style utility if installed:

```
nc 127.0.0.1 5555
```

Type text and verify that the bytes are echoed back. Open multiple clients simultaneously to observe that the server continues accepting and serving independent sessions.

3.8 Final reusable pattern

3.8.1 The pattern in one sentence

A session-based Asio server accepts a connected socket, transfers ownership of that socket into a shared session object, starts the first asynchronous operation, and keeps the session alive by capturing `shared_from_this()` in every handler that depends on the object.

3.8.2 The pattern in compact form

1. the server owns the acceptor,
2. the server starts `async_accept()`,
3. on accept, it constructs `std::make_shared<session>(std::move(socket))`,
4. it calls `start()`,

5. every asynchronous session step captures `self = shared_from_this()`,
6. buffers used by outstanding operations remain alive and stable,
7. when no more work is issued and no ownership remains, the session is destroyed naturally.

3.8.3 What to remember most

For real Boost.Asio code, the hardest errors are often not syntax errors and not even networking errors. They are ownership and lifetime errors. The session-based design, combined with the `shared_from_this()` pattern and stable buffer ownership, is the essential discipline that makes asynchronous TCP code reliable.

3.9 Key takeaways

- A session object is the natural owner of one connected client.
- `start()` usually begins the first asynchronous step.
- `shared_from_this()` keeps the session alive across handlers.
- Buffers must stay alive and remain stable until completion.
- The most reusable design separates listener logic from per-client session logic.

Reading Data Correctly

4.1 Introduction

Reading from a TCP socket is the point where many first backend servers become subtly wrong. The code may compile, accept clients, and even appear to work in quick tests, yet still contain protocol bugs. The reason is simple: TCP is a byte stream, not a message queue. A read operation reports that some bytes were received, not that a complete logical application message has arrived.

For correct server design, the programmer must distinguish between the transport layer and the application protocol:

1. TCP guarantees ordered byte delivery.
2. TCP does not preserve application message boundaries.
3. A single logical message may arrive in several reads.
4. Several logical messages may arrive in one read.
5. Buffer ownership and buffer lifetime must remain correct for the entire asynchronous operation.

This chapter focuses on those realities and on the Boost.Asio facilities that exist specifically to handle them correctly.

4.2 Partial reads

4.2.1 The core rule

A call such as `socket.async_read_some(...)` does not mean “read one full application message”. It means “read some bytes that are currently available, up to the size of the supplied buffer”. This distinction is essential.
If the protocol message is 100 bytes long, the first read may return:

- 100 bytes,
- 60 bytes,
- 12 bytes,
- or even several complete messages plus part of another, if the peer has already transmitted enough data and the buffer is large enough.

Therefore, application code must define how message boundaries are recognized.

4.2.2 Why `async_read_some()` is low-level

The member function `async_read_some()` is the low-level socket read primitive. It is ideal when the programmer explicitly wants chunk-oriented stream processing. It is not, by itself, a protocol parser.
A common first mistake is to assume that this code reads one full command:

```
socket_.async_read_some(
    boost::asio::buffer(data_),
    [self](boost::system::error_code ec, std::size_t n)
    {
```

```
        if (!ec)
        {
            std::string command(self->data_.data(), n);
            self->handle_command(command);
        }
    });
```

This is only correct if the protocol is explicitly defined as “whatever bytes happened to arrive in one read”, which is rarely the case.

4.2.3 Typical failure modes

Ignoring partial reads usually leads to one of the following bugs:

1. a long line-based command is truncated and parsed too early,
2. a binary header is only partially received and is interpreted as complete,
3. two back-to-back messages are merged and parsed as one,
4. leftover bytes from one message are accidentally discarded before the next parse step.

4.2.4 Correct design choices

In practice, the application should choose one of the following protocol strategies:

- fixed-size records,
- delimiter-based records such as `"\n"` or `"\r\n"`,
- length-prefixed binary frames,
- grammar-driven parsing using a persistent input buffer.

Each strategy implies a different reading pattern.

4.2.5 Using `async_read()` when the size is known

If the program knows exactly how many bytes are required before parsing may proceed, the composed operation `boost::asio::async_read()` is often the correct tool. It continues internally through zero or more calls to `async_read_some()` until the supplied buffers are full or an error occurs.
This makes it ideal for:

- fixed-size binary headers,
- fixed-size records,
- exact reads into a structure-compatible byte array,
- reading a known payload length after a header has already been parsed.

4.2.6 Exact-size read example

The following example shows a session that reads exactly 8 bytes before processing the result:

```
#include <array>
#include <iostream>
#include <memory>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class fixed_header_session
    : public std::enable_shared_from_this<fixed_header_session>
```

```
{
public:
    explicit fixed_header_session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        read_header();
    }

private:
    void read_header()
    {
        auto self = shared_from_this();

        asio::async_read(
            socket_,
            asio::buffer(header_),
            [self](error_code ec, std::size_t bytes_transferred)
            {
                if (ec)
                {
                    std::cerr << "read header error: "
                              << ec.message() << '\n';
                    return;
                }

```

```
                std::cout << "header bytes read: "
                          << bytes_transferred << '\n';

                self->process_header();
            });
    }

    void process_header()
    {
        std::cout << "8-byte header is complete and ready to parse\n";
    }

private:
    tcp::socket socket_;
    std::array<unsigned char, 8> header_{};
};
```

4.2.7 Why this works

The previous code does not assume that one low-level read returns all 8 bytes. Instead, it delegates that work to the composed operation. This is exactly the correct mental model:

> Use `async_read_some()` for raw chunk handling. Use `async_read()` when the application needs an exact amount of data before parsing.

4.2.8 Do not overlap reads

A very important rule is that a stream should not have overlapping read operations of the same direction in progress. If a composed read is active, the program must not simultaneously issue another read on the same socket.

The safest design is simple: one active read chain per session, where each completion handler decides what the next read should be.

4.3 Buffer handling

4.3.1 What a Boost.Asio buffer really is

A Boost.Asio buffer object does not own memory. It is only a lightweight descriptor that refers to an existing memory region by pointer and size. This design is efficient, but it means the application remains responsible for validity of the underlying storage.
This is one of the most important correctness rules in all asynchronous networking code.

4.3.2 Practical consequence

When the program writes:

```
asio::buffer(data_)
```

the program is not creating new storage. It is merely creating a view onto existing storage. Therefore:

- the referenced memory must remain alive until the operation completes,
- the bytes must remain stable if the operation depends on their contents,
- any container reallocation or invalidation can silently break correctness.

4.3.3 Safe buffer sources

For session-oriented code, the most practical safe buffer sources are:

- `std::array<char, N>` member buffers,
- `std::vector<unsigned char>` with stable lifetime and careful resizing rules,
- `std::string` for textual protocol data,
- dynamic buffers such as `boost::asio::streambuf` or `boost::asio::dynamic_buffer(std::string)`.

4.3.4 Static member buffer example

A member array is one of the simplest correct read buffers:

```
class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        do_read();
    }

private:
    void do_read()
    {
        auto self = shared_from_this();

```

```
        socket_.async_read_some(
            asio::buffer(data_),
            [self](error_code ec, std::size_t n)
            {
                if (ec)
                    return;

                self->handle_chunk(n);
                self->do_read();
            });
    }

    void handle_chunk(std::size_t n)
    {
        std::cout << "received chunk of " << n << " bytes\n";
    }

private:
    tcp::socket socket_;
    std::array<char, 4096> data_{};
};
```

This works because the array is a member of the session object, and the session object remains alive through `shared_from_this()`.

4.3.5 Vector example and its pitfall

A vector can also be used:

```
std::vector<unsigned char> data_{4096};
```

```

socket_.async_read_some(
    asio::buffer(data_),
    [self](error_code ec, std::size_t n)
    {
        if (!ec)
            self->handle_chunk(n);
    });
```

However, the program must not resize or otherwise invalidate the vector's storage while the read is outstanding. The buffer size is based on the vector's `size()`, not its capacity.

4.3.6 Incorrect temporary buffer example

The following is wrong:

```
void do_read_bad()
{
    std::array<char, 1024> temp{};

    socket_.async_read_some(
        asio::buffer(temp),
        [](error_code, std::size_t)
        {
        });
}
```

The array dies when the function returns, but the asynchronous operation may complete later.

4.3.7 String buffer example

For text-oriented protocols, a string can be convenient:

```
std::string text_buffer(1024, '\0');

socket_.async_read_some(
    asio::buffer(text_buffer),
    [self](error_code ec, std::size_t n)
    {
        if (!ec)
        {
            std::string_view chunk(self->text_buffer.data(), n);
            std::cout << "text chunk: " << chunk << '\n';
        }
    });
```

Again, the string must not be invalidated during the outstanding operation.

4.3.8 Buffer arithmetic

Boost.Asio buffer objects support safe slicing through offset arithmetic. This is useful when only part of a buffer remains to be filled.

```
std::array<char, 16> packet{};
std::size_t already_have = 5;

socket_.async_read_some(
    asio::buffer(asio::buffer(packet) + already_have),
    [self](error_code ec, std::size_t n)
    {
        if (!ec)
        {
            // n bytes were placed starting at offset already_have
```

```
        }
    });
```

This technique is useful for hand-built state machines, though for exact-size reads `async_read()` is often simpler.

4.3.9 Scatter-gather buffers

Boost.Asio also supports reading or writing across multiple buffers in one operation. This is useful for:

- separate header and body regions,
- fixed structure prefixes plus payload areas,
- protocol framing without copying into one contiguous temporary object.

Example:

```
std::array<unsigned char, 4> header{};
std::array<unsigned char, 1024> payload{};

std::array<asio::mutable_buffer, 2> bufs = {
    asio::buffer(header),
    asio::buffer(payload, 128)
};

socket_.async_read_some(
    bufs,
    [self](error_code ec, std::size_t n)
    {
```

```
13          if (!ec)
14              std::cout << "received across multiple buffers: "
15                        << n << " bytes\n";
16      });
```

4.3.10 Dynamic buffers

For protocols where the incoming logical record size is not known in advance, dynamic buffers are often the best fit. Two very practical choices are:

- `boost::asio::streambuf`,
- `boost::asio::dynamic_buffer(std::string)`.

These are particularly useful with `read_until()` and `async_read_until()`.

4.3.11 Why dynamic buffers matter

A delimiter-based protocol needs somewhere to accumulate bytes until the delimiter is found. A fixed-size stack buffer is not sufficient by itself, because the data may arrive in several chunks and may include extra bytes belonging to subsequent messages.
A dynamic buffer solves this by preserving all received data that has not yet been consumed.

4.4 Line-based protocol example

4.4.1 Why line-based reading is special

Many application protocols are line-oriented. Common delimiters include:

- `"\n"`,

- `"\r\n"`.

For this case, the correct Asio tool is often `async_read_until()`. It reads into a dynamic buffer until the delimiter has appeared in the readable region.
This is better than manual chunk assembly for most simple command protocols.

4.4.2 Important behavior of `async_read_until()`

The function does not mean “read exactly one line and discard the rest”. It means:

1. keep reading until the delimiter is found in the dynamic buffer,
2. report how many bytes belong to the matched record up to and including the delimiter,
3. leave any extra bytes after the delimiter in the dynamic buffer for later parsing.

That last property is extremely important and is one of the main reasons line-based parsing works correctly with this function.

4.4.3 Line-based command session using `streambuf`

The following reusable session reads one line at a time and echoes a processed response.

```
#include <iostream>
#include <istream>
#include <memory>
#include <string>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
```

```
using boost::system::error_code;

class line_session : public std::enable_shared_from_this<line_session>
{
public:
    explicit line_session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        read_line();
    }

private:
    void read_line()
    {
        auto self = shared_from_this();

        asio::async_read_until(
            socket_,
            input_,
            '\n',
            [self](error_code ec, std::size_t bytes_transferred)
            {
                if (ec)
                {
                    if (ec != asio::error::eof)
```

```
                {
                    std::cerr << "read line error: "
                              << ec.message() << '\n';
                }
                return;
            }

            self->handle_line(bytes_transferred);
        });
    }

    void handle_line(std::size_t bytes_transferred)
    {
        std::istream is(&input_);
        std::string line;
        std::getline(is, line);

        if (!line.empty() && line.back() == '\r')
            line.pop_back();

        std::cout << "line received (" << bytes_transferred
                  << " bytes including delimiter): "
                  << line << '\n';

        output_ = "OK: " + line + "\n";
        write_response();
    }

    void write_response()
```

```
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(output_),
            [self](error_code ec, std::size_t)
            {
                if (ec)
                {
                    std::cerr << "write response error: "
                              << ec.message() << '\n';
                    return;
                }

                self->read_line();
            });
    }

private:
    tcp::socket socket_;
    asio::streambuf input_;
    std::string output_;
};
```

4.4.4 Why this code is correct

This line-based session is correct for several reasons:

1. the dynamic buffer persists across reads,

2. the session reads until a delimiter, not until an arbitrary chunk boundary,
3. any surplus bytes after the first delimiter remain in the stream buffer,
4. the response string remains alive until the asynchronous write completes.

4.4.5 Using `dynamic_buffer(std::string)` instead

A modern alternative is to accumulate into a string-backed dynamic buffer:

```
class line_session_string
    : public std::enable_shared_from_this<line_session_string>
{
public:
    explicit line_session_string(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        read_line();
    }

private:
    void read_line()
    {
        auto self = shared_from_this();

        asio::async_read_until(
            socket_,
```

```
            asio::dynamic_buffer(input_),
            "\r\n",
            [self](error_code ec, std::size_t bytes_transferred)
            {
                if (ec)
                    return;

                std::string line = self->input_.substr(0, bytes_transferred);

                if (line.size() >= 2 &&
                    line[line.size() - 2] == '\r' &&
                    line[line.size() - 1] == '\n')
                {
                    line.resize(line.size() - 2);
                }

                self->input_.erase(0, bytes_transferred);

                self->output_ = "ECHO: " + line + "\r\n";
                self->write_response();
            });
    }

    void write_response()
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
```

```
            asio::buffer(output_),
            [self](error_code ec, std::size_t)
            {
                if (!ec)
                    self->read_line();
            });
    }

private:
    tcp::socket socket_;
    std::string input_;
    std::string output_;
};
```

4.4.6 When to prefer line-based reading

A delimiter-based design is especially suitable for:

- small custom text commands,
- REPL-style protocol sessions,
- lightweight admin ports,
- SMTP-like, FTP-like, or HTTP-like control lines.

It is usually not the best choice for arbitrary binary payloads.

4.5 Binary protocol example

4.5.1 Why binary protocols need explicit framing

A binary protocol cannot safely rely on delimiter search unless its framing was specifically designed that way. In most cases, binary protocols use one of these patterns:

- fixed-size records,
- fixed-size header plus variable-size payload,
- tag-length-value structures,
- packet length prefix.

The most common and reusable design is a fixed-size header that includes the payload length.

4.5.2 Length-prefixed protocol strategy

A very typical binary frame looks like this:

1. first 4 bytes: payload length in network byte order,
2. next N bytes: payload.

This makes the reading algorithm straightforward:

1. read exactly 4 bytes,
2. decode the length,
3. validate it against a maximum,
4. resize storage for the payload,

5. read exactly that many bytes,

6. process the complete message.

4.5.3 Why this pattern is robust

This pattern works well because it never guesses where a message ends. It explicitly learns the payload size from the protocol itself and then uses exact-size reads.

4.5.4 Complete binary session example

The following session implements a reusable length-prefixed binary reader.

```
#include <array>
#include <cstdint>
#include <iostream>
#include <memory>
#include <vector>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class binary_session : public std::enable_shared_from_this<binary_session>
{
public:
    explicit binary_session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }
```

```
19
20      void start()
21      {
22          read_header();
23      }
24
25  private:
26      static std::uint32_t load_be32(const std::array<unsigned char, 4>& b)
27      {
28          return (static_cast<std::uint32_t>(b[0]) << 24) |
29                 (static_cast<std::uint32_t>(b[1]) << 16) |
30                 (static_cast<std::uint32_t>(b[2]) << 8)  |
31                 (static_cast<std::uint32_t>(b[3]));
32      }
33
34      static void store_be32(std::uint32_t value, std::array<unsigned char, 4>& b)
35      {
36          b[0] = static_cast<unsigned char>((value >> 24) & 0xFF);
37          b[1] = static_cast<unsigned char>((value >> 16) & 0xFF);
38          b[2] = static_cast<unsigned char>((value >> 8) & 0xFF);
39          b[3] = static_cast<unsigned char>(value & 0xFF);
40      }
41
42      void read_header()
43      {
44          auto self = shared_from_this();
45
46          asio::async_read(
47              socket_,
```

```
48              asio::buffer(header_),
49              [self](error_code ec, std::size_t)
50              {
51                  if (ec)
52                  {
53                      std::cerr << "read header error: "
54                                << ec.message() << '\n';
55                      return;
56                  }
57
58                  self->payload_length_ = load_be32(self->header_);
59
60                  constexpr std::uint32_t max_payload = 1024 * 1024;
61                  if (self->payload_length_ > max_payload)
62                  {
63                      std::cerr << "payload too large: "
64                                << self->payload_length_ << '\n';
65                      return;
66                  }
67
68                  self->payload_.assign(self->payload_length_, 0);
69                  self->read_payload();
70              });
71      }
72
73      void read_payload()
74      {
75          auto self = shared_from_this();
76
```

```
        if (payload_.empty())
        {
            process_message();
            return;
        }

        asio::async_read(
            socket_,
            asio::buffer(payload_),
            [self](error_code ec, std::size_t)
            {
                if (ec)
                {
                    std::cerr << "read payload error: "
                              << ec.message() << '\n';
                    return;
                }

                self->process_message();
            });
    }

    void process_message()
    {
        std::cout << "binary message complete, payload bytes: "
                  << payload_.size() << '\n';

        response_body_ = payload_;

```

```
        std::array<unsigned char, 4> response_header{};
        store_be32(static_cast<std::uint32_t>(response_body_.size()),
                   response_header);

        response_frame_.clear();
        response_frame_.insert(response_frame_.end(),
                               response_header.begin(),
                               response_header.end());
        response_frame_.insert(response_frame_.end(),
                               response_body_.begin(),
                               response_body_.end());

        write_response();
    }

    void write_response()
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(response_frame_),
            [self](error_code ec, std::size_t)
            {
                if (ec)
                {
                    std::cerr << "write response error: "
                              << ec.message() << '\n';
                    return;
```

```
                }

                self->read_header();
            });
    }

private:
    tcp::socket socket_;
    std::array<unsigned char, 4> header_{};
    std::uint32_t payload_length_ = 0;
    std::vector<unsigned char> payload_;
    std::vector<unsigned char> response_body_;
    std::vector<unsigned char> response_frame_;
};
```

4.5.5 Why this binary reader is correct

This design is correct because:

1. the header read uses `async_read()` to obtain exactly 4 bytes,
2. the payload length is parsed only after the header is complete,
3. the payload buffer is resized before the exact-size payload read begins,
4. the response frame remains stable until `async_write()` completes,
5. the session restarts at header-read state after each successful message.

4.5.6 Validation matters

A binary protocol must validate sizes before allocating memory. Without a maximum payload limit, a malicious peer could announce an excessively large frame and force uncontrolled allocation attempts.
Therefore, every real binary protocol should define:

- maximum frame size,
- legal field ranges,
- timeout policy for incomplete frames,
- behavior for malformed input.

4.5.7 Alternative binary pattern: fixed-size record

For very small protocols, a fixed-size record may be simpler than a length prefix.
Example: a 12-byte command record.

```
std::array<unsigned char, 12> record_{};

asio::async_read(
    socket_,
    asio::buffer(record_),
    [self](error_code ec, std::size_t)
    {
        if (!ec)
        {
            self->process_fixed_record();
            self->read_next_record();
```

```
        }
    });
```

This works only when the protocol truly guarantees fixed-size messages.

4.6 Choosing the right read model

4.6.1 A practical selection guide

The most useful mental rule is:

- use `async_read_some()` when you want raw transport chunks,
- use `async_read()` when you know exactly how many bytes are required,
- use `async_read_until()` when protocol framing is delimiter-based,
- keep unconsumed bytes in a persistent buffer whenever parsing may need leftovers.

4.6.2 Do not discard surplus bytes

A frequent beginner error is to parse one logical message and then clear the entire buffer. This is wrong if extra bytes already belong to the next message.
For delimiter-based protocols, extra bytes after the matched delimiter must remain available for the next parse step.

4.6.3 Read path architecture

A robust server session usually follows this shape:

1. maintain one persistent input buffer per connection,

2. issue exactly one active read chain at a time,
3. parse only when enough data for one logical record exists,
4. consume only the bytes that belong to that record,
5. preserve all remaining bytes for the next step.

That architecture works for text protocols, binary protocols, and mixed framing designs.

4.7 Windows build guidance

4.7.1 Typical MSVC compile command

Assuming Boost headers are available at:

```
C:\Libraries\boost_1_89_0
```

a typical Developer Command Prompt build is:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 reading_data_correctly.cpp
```

4.7.2 Typical MinGW-w64 compile command

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0
↪  reading_data_correctly.cpp -o reading_data_correctly.exe
```

4.7.3 Header-only note

For the examples in this chapter, normal header-only Boost.Asio usage is sufficient.

4.8 Final reusable pattern

4.8.1 The pattern in one sentence

Correct network reading means preserving input state across operations, choosing a framing strategy that matches the protocol, and ensuring that every buffer referenced by an outstanding operation remains valid and stable until completion.

4.8.2 Reusable read-loop template

The following template summarizes the most practical design:

```
class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        read_next_message();
    }

private:
    void read_next_message()
    {
        // Choose exactly one:
        // 1. async_read_some(...)       for raw chunks
```

```
        // 2. async_read(...)            for exact-size fields
        // 3. async_read_until(...)      for delimiter-based records
    }

    void process_message()
    {
        // Parse only complete logical records.
        // Preserve leftover bytes if the protocol allows them.
    }

private:
    tcp::socket socket_;

    // One or more persistent input buffers:
    std::array<char, 4096> chunk_buffer_{};
    boost::asio::streambuf line_buffer_;
    std::vector<unsigned char> binary_payload_;
    std::string text_storage_;
};
```

4.9 Key takeaways

- TCP delivers ordered bytes, not application messages.
- `async_read_some()` may return partial or merged logical records.
- `async_read()` is the right tool for exact-size reads.
- `async_read_until()` is the right tool for delimiter-based protocols.
- Boost.Asio buffers do not own memory; lifetime remains the program's responsibility.

- Unconsumed bytes must be preserved when the protocol requires them.

Writing Data Correctly

5.1 Introduction

Reading correctness is only half of network correctness. A server that reads properly but writes carelessly will still fail under realistic workloads. In practice, write-side bugs are among the most common problems in first Boost.Asio servers, especially when several responses may be produced close together or from different parts of the program.
The write side has four core realities:

1. a socket write may be partial at the low level,
2. `boost::asio::async_write()` is the usual composed operation when the intention is to send an entire buffer sequence,
3. the memory referenced by the buffers must remain alive and stable until completion,
4. overlapping writes on the same stream require deliberate serialization.

This chapter focuses on the modern and correct patterns for sending data through a TCP session on Windows with Boost.Asio.

5.2 `async_write` usage

5.2.1 What `async_write` actually does

The function `boost::asio::async_write()` is a composed asynchronous operation. It does not directly correspond to one operating system write call. Instead, it repeatedly performs lower-level write steps until one of the following becomes true:

1. all supplied bytes have been written,
2. an error occurs.

This makes it different from a single low-level send-style operation. A low-level operation may write only part of the requested data. By contrast, `async_write()` exists specifically for the common case where application code wants to send the full supplied buffer sequence.

5.2.2 Why this matters

A common beginner assumption is that one write call always sends the entire message. That assumption is not safe at the transport layer. TCP is a byte stream, and low-level write operations may complete after sending only part of the requested data.
Therefore, when the requirement is:

> send this complete response before considering the write finished

the usual tool is:

```
boost::asio::async_write(...)
```

rather than a single `async_write_some()` call.

5.2.3 Minimal example

The following example sends a full textual response correctly:

```
#include <iostream>
#include <memory>
#include <string>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using boost::system::error_code;
using asio::ip::tcp;

class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        send_greeting();
    }

private:
    void send_greeting()
    {
        auto self = shared_from_this();
```

```
        auto msg = std::make_shared<std::string>(
            "Hello from Boost.Asio server\r\n");

        asio::async_write(
            socket_,
            asio::buffer(*msg),
            [self, msg](error_code ec, std::size_t bytes_transferred)
            {
                if (ec)
                {
                    std::cerr << "write error: "
                              << ec.message() << '\n';
                    return;
                }

                std::cout << "sent "
                          << bytes_transferred
                          << " bytes\n";
            });
    }

private:
    tcp::socket socket_;
};
```

5.2.4 Why the message uses shared ownership

The string object referenced by `asio::buffer(*msg)` must remain valid until the completion handler runs. The buffer object itself does not own the string. It is only a lightweight view over

existing storage.

This is why the message is stored in a `std::shared_ptr<std::string>` and captured by value in the lambda. If the message were a local string variable, it would be destroyed when the function returned, while the asynchronous operation might still be in progress.

5.2.5 Incorrect temporary-buffer example

The following code is wrong:

```
void send_bad()
{
    std::string msg = "This looks fine but is unsafe\r\n";

    asio::async_write(
        socket_,
        asio::buffer(msg),
        [](boost::system::error_code, std::size_t)
        {
        });
}
```

The string `msg` is destroyed when `send_bad()` returns. The asynchronous write may complete later, which means the buffer refers to invalid memory.

5.2.6 Correct member-buffer example

Another correct approach is to keep the outgoing bytes inside the session object:

```
class session : public std::enable_shared_from_this<session>
{
public:
```

```
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        outgoing_ = "Persistent member-backed message\r\n";
        do_write();
    }

private:
    void do_write()
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(outgoing_),
            [self](boost::system::error_code ec, std::size_t)
            {
                if (ec)
                    return;

                self->outgoing_.clear();
            });
    }

private:
```

```
    tcp::socket socket_;
    std::string outgoing_;
};
```

This is safe because `outgoing_` belongs to the session object, and the session object remains alive due to `shared_from_this()`.

5.2.7 Using buffer sequences

`async_write()` can also send multiple buffers as one logical write operation. This is useful when the message is naturally split into parts such as header and payload.

```
std::string header = "LEN=5\r\n";
std::string body   = "Hello";

std::array<asio::const_buffer, 2> bufs = {
    asio::buffer(header),
    asio::buffer(body)
};

asio::async_write(
    socket_,
    bufs,
    [self, header = std::move(header), body = std::move(body)]
    (boost::system::error_code ec, std::size_t bytes_transferred)
    {
        if (!ec)
        {
            std::cout << "sent "
                      << bytes_transferred
```

```
                    << " bytes\n";
        }
    });
```

The important detail is still the same: the underlying storage for all buffers must remain valid until completion.

5.2.8 When to use `async_write_some`

The lower-level operation `async_write_some()` is appropriate when the programmer explicitly wants manual control over partial writes and intends to manage the remaining bytes directly. That is useful in custom composed operations, special streaming designs, and advanced protocols.
For most session-level response sending, however, `async_write()` is simpler and more obviously correct.

5.3 Write queue pattern

5.3.1 Why a queue is needed

A real server rarely sends only one message in isolation. More commonly:

- a command handler wants to send a response,
- a timer wants to send a notification,
- protocol logic wants to send an acknowledgement,
- several events may request writes close together.

If each of those code paths starts its own asynchronous write immediately, the session may accidentally initiate overlapping writes on the same socket. That is one of the most common write-side correctness bugs.
The standard solution is a per-session write queue.

5.3.2 Core idea of the queue

The queue pattern is simple:

1. every outgoing message is appended to a queue,
2. if no write is currently active, start one write using the front element,
3. when that write completes, remove the front element,
4. if more messages remain, start the next write,
5. at all times, at most one write operation is active for that session.

This pattern is used repeatedly in classic asynchronous chat-style and session-style designs because it is robust, readable, and easy to extend.

5.3.3 Minimal queue-based writer

The following session demonstrates the essential queue pattern:

```
#include <deque>
#include <iostream>
#include <memory>
#include <string>
#include <boost/asio.hpp>

```

```
namespace asio = boost::asio;
using boost::system::error_code;
using asio::ip::tcp;

class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void deliver(std::string msg)
    {
        bool write_in_progress = !write_queue_.empty();
        write_queue_.push_back(std::move(msg));

        if (!write_in_progress)
            do_write();
    }

private:
    void do_write()
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(write_queue_.front()),
```

```
            [self](error_code ec, std::size_t bytes_transferred)
            {
                if (ec)
                {
                    std::cerr << "write error: "
                              << ec.message() << '\n';
                    return;
                }

                std::cout << "sent "
                          << bytes_transferred
                          << " bytes\n";

                self->write_queue_.pop_front();

                if (!self->write_queue_.empty())
                    self->do_write();
            });
    }

private:
    tcp::socket socket_;
    std::deque<std::string> write_queue_;
};
```

5.3.4 Why this works

This pattern works for two important reasons:

1. the front queue element remains alive during the outstanding write,

2. the next write is not initiated until the previous write has completed.

Therefore both buffer lifetime and write sequencing remain correct.

5.3.5 Why `std::deque` is convenient

A `std::deque<std::string>` is a practical default for queued outgoing messages:

- it preserves clear FIFO semantics,
- pushing to the back and popping from the front are natural operations,
- each string owns its own stable character storage,
- the code remains simple and explicit.

Other containers can work, but `std::deque` is often the clearest choice.

5.3.6 Do not overwrite one member string repeatedly

A subtle mistake is to reuse a single member string for several outgoing writes:

```
void send_twice_bad()
{
    output_ = "first message";

    asio::async_write(
        socket_,
        asio::buffer(output_),
        [](boost::system::error_code, std::size_t) {});

    output_ = "second message";
}
```

Even though `output_` still exists, its contents have changed before the first write necessarily completed. The queue pattern avoids this by giving each pending message its own stable storage.

5.3.7 Queue plus member function interface

A practical public interface is usually a method such as:

```
void deliver(std::string msg);
```

This hides the internal queue and gives other parts of the server a clean, safe way to request output.

5.4 Avoiding race conditions

5.4.1 Where races appear

In a single-threaded `io_context::run()` design, many first servers are naturally protected from certain concurrent handler execution issues because only one thread is pumping the context. Even then, logical overlap bugs can still happen if several code paths initiate writes without sequencing.
In a multi-threaded design, the risk becomes greater:

- multiple threads may call `io.run()`,
- handlers for the same session may execute on different worker threads,
- unsynchronized access to the write queue becomes unsafe,
- starting concurrent writes from different handlers becomes easier by accident.

5.4.2 Two distinct problems

It is useful to separate two different problems:

1. transport sequencing: do not overlap writes on the same socket unless the design explicitly allows and manages it,
2. shared-state synchronization: do not let multiple handlers mutate the same queue or session state concurrently without serialization.

A correct writer must solve both.

5.4.3 Single-threaded chain is not the whole story

A read chain often works naturally because the code starts the next read only from the current read handler. This creates an implicit sequence.
Write requests are different. They often originate from many different places. For example:

- a parsed command handler,
- a timer callback,
- a broadcast dispatcher,
- a business-logic completion path.

Therefore write sequencing usually needs a deliberate design instead of relying on accidental control flow.

5.4.4 Using a strand

A standard Boost.Asio solution is to serialize a session's handlers through a strand. The idea is not to make the program single-threaded globally. Instead, it ensures that handlers associated with the same session serialization domain do not execute concurrently.
A practical session member is:

```
asio::strand<asio::io_context::executor_type> strand_;
```

or, in a more generic modern style based on the socket's executor:

```
asio::strand<asio::any_io_executor> strand_;
```

The strand becomes the serialization mechanism for queue updates and write initiation.

5.4.5 Posting into the strand

A common safe pattern is to make the public `deliver()` function post its queue work into the session strand:

```
void deliver(std::string msg)
{
    auto self = shared_from_this();

    asio::post(
        strand_,
        [self, msg = std::move(msg)]() mutable
        {
            bool write_in_progress = !self->write_queue_.empty();
            self->write_queue_.push_back(std::move(msg));

```

```
            if (!write_in_progress)
                self->do_write();
        });
}
```

This has two important advantages:

1. queue mutation is serialized,
2. the decision whether to start a new write is serialized with the queue mutation.

5.4.6 Binding write handlers to the same strand

The write completion handler should also execute in the same serialization domain. A practical way is to ensure the socket operations themselves use the strand-associated executor or explicitly bind handlers to the strand.
If the session socket and associated handlers are consistently launched from the strand, then queue mutation, completion handling, and write chaining remain serialized.

5.4.7 Unsafe multi-threaded example

The following shape is unsafe in a multi-threaded server:

```
void deliver_unsafe(std::string msg)
{
    bool write_in_progress = !write_queue_.empty();
    write_queue_.push_back(std::move(msg));

    if (!write_in_progress)
        do_write();
}
```

If two threads call this concurrently, several failures are possible:

- data race on `write_queue_`,
- duplicate calls to `do_write()`,
- corrupted sequencing assumptions,
- overlapping writes on the same socket.

5.4.8 Why a mutex is not the first Asio answer

A mutex can be made to work, but in Asio-style code it is often better to use executor-based serialization for per-session asynchronous state. The strand fits naturally into the event-driven model and keeps state changes in the same execution framework as the socket handlers themselves.

5.4.9 Race-free write rule

A very practical rule is:

> For each session, all write-queue mutation and write-chain continuation must happen through one serialization domain.

That domain may be:

- one single `io_context` thread,
- one strand inside a multi-threaded `io_context`,
- or another carefully designed synchronization mechanism.

For most Boost.Asio session code, the strand is the clearest answer.

5.5 Final safe writer implementation

5.5.1 Design goals

A reusable session writer should satisfy all of the following:

1. all outgoing messages are fully written using `async_write()`,
2. no overlapping writes occur on one socket,
3. the queue remains safe under multi-threaded `io_context::run()` execution,
4. outgoing buffers remain alive and stable until completion,
5. the public interface is easy to call from protocol logic.

5.5.2 Complete implementation

The following session class is a practical safe writer foundation for Windows server development:

```
#include <deque>
#include <iostream>
#include <memory>
#include <string>
#include <utility>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using boost::system::error_code;
using asio::ip::tcp;

```

```
class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket)),
          strand_(socket_.get_executor())
    {
    }

    void start()
    {
        std::cout << "writer-ready session started\n";
    }

    void deliver(std::string msg)
    {
        auto self = shared_from_this();

        asio::post(
            strand_,
            [self, msg = std::move(msg)]() mutable
            {
                bool write_in_progress = !self->write_queue_.empty();
                self->write_queue_.push_back(std::move(msg));

                if (!write_in_progress)
                    self->do_write();
            });
    }
```

```
41
42      void close()
43      {
44          auto self = shared_from_this();
45
46          asio::post(
47              strand_,
48              [self]()
49              {
50                  error_code ignored_ec;
51                  self->socket_.shutdown(tcp::socket::shutdown_both, ignored_ec);
52                  self->socket_.close(ignored_ec);
53              });
54      }
55
56  private:
57      void do_write()
58      {
59          auto self = shared_from_this();
60
61          asio::async_write(
62              socket_,
63              asio::buffer(write_queue_.front()),
64              asio::bind_executor(
65                  strand_,
66                  [self](error_code ec, std::size_t bytes_transferred)
67                  {
68                      if (ec)
69                      {
```

```
                        std::cerr << "write error: "
                                  << ec.message() << '\n';
                        return;
                    }

                    std::cout << "sent "
                              << bytes_transferred
                              << " bytes\n";

                    self->write_queue_.pop_front();

                    if (!self->write_queue_.empty())
                        self->do_write();
                }));
    }

private:
    tcp::socket socket_;
    asio::strand<asio::any_io_executor> strand_;
    std::deque<std::string> write_queue_;
};
```

5.5.3 Why this implementation is safe

This implementation is safe for the intended session-writing role because:

1. every public write request is funneled through the strand,
2. queue mutation and the idle-or-busy decision happen in the same serialized context,
3. only one active write is started at a time,

4. the front queued string remains alive while it is being written,
5. the completion handler also runs in the same strand,
6. the next write is started only after the previous one has completed.

5.5.4 Why the front queue element is a valid buffer source

The expression:

```
asio::buffer(write_queue_.front())
```

is safe because the front string object remains inside the queue until the completion handler calls:

```
write_queue_.pop_front();
```

Thus the buffer refers to stable storage for the full duration of the outstanding operation.

5.5.5 What happens on error

In the example above, a write error ends the write chain by returning from the handler. That is a sensible minimal design because once the transport is broken, continuing the queue usually has no meaning.
In a richer server, the handler may additionally:

- notify higher-level session logic,
- log the remote endpoint,
- cancel timers,
- initiate cleanup,
- remove the session from a connection registry.

5.5.6 Supporting binary payloads

The same queue pattern works for binary messages. One practical variant is:

```
using message_type = std::vector<unsigned char>;
std::deque<message_type> write_queue_;
```

Then each queued vector owns stable byte storage until completion. The logic of `deliver()` and `do_write()` remains essentially the same.

5.5.7 Supporting framed responses

For framed protocols, it is often best to queue the fully encoded frame rather than separate fragments. For example, if the protocol is length-prefixed, build the entire header-plus-body frame first, then push that frame into the queue as one message object. This reduces lifetime complexity and keeps the writer generic.

5.5.8 Broadcast-style usage

If a server broadcasts one logical message to many sessions, each session should usually enqueue its own owned copy or otherwise share an immutable message object whose storage is guaranteed to remain valid until every session has completed its write. The key principle is unchanged: each outstanding write must reference stable memory.

5.6 A complete echo-style example with safe writing

5.6.1 Session with read loop and safe writer

The following example combines a simple read path with the safe writer queue. Each received line is echoed back by enqueuing a response through `deliver()`.

```cpp
#include <array>
#include <deque>
#include <iostream>
#include <memory>
#include <string>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using boost::system::error_code;
using asio::ip::tcp;

class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket)),
          strand_(socket_.get_executor())
    {
    }

    void start()
    {
        do_read();
    }

private:
    void do_read()
    {
        auto self = shared_from_this();
```

```
30
31          socket_.async_read_some(
32              asio::buffer(data_),
33              asio::bind_executor(
34                  strand_,
35                  [self](error_code ec, std::size_t length)
36                  {
37                      if (ec)
38                      {
39                          if (ec != asio::error::eof)
40                          {
41                              std::cerr << "read error: "
42                                        << ec.message() << '\n';
43                          }
44                          return;
45                      }
46
47                      std::string msg(self->data_.data(), length);
48                      self->deliver("ECHO: " + msg);
49                      self->do_read();
50                  }));
51      }
52
53      void deliver(std::string msg)
54      {
55          auto self = shared_from_this();
56
57          asio::post(
58              strand_,
```

```
            [self, msg = std::move(msg)]() mutable
            {
                bool write_in_progress = !self->write_queue_.empty();
                self->write_queue_.push_back(std::move(msg));

                if (!write_in_progress)
                    self->do_write();
            });
    }

    void do_write()
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(write_queue_.front()),
            asio::bind_executor(
                strand_,
                [self](error_code ec, std::size_t)
                {
                    if (ec)
                    {
                        std::cerr << "write error: "
                                  << ec.message() << '\n';
                        return;
                    }

                    self->write_queue_.pop_front();
```

```
                    if (!self->write_queue_.empty())
                        self->do_write();
                }));
    }

private:
    tcp::socket socket_;
    asio::strand<asio::any_io_executor> strand_;
    std::array<char, 4096> data_{};
    std::deque<std::string> write_queue_;
};
```

5.6.2 What this example demonstrates

This session demonstrates the most important writing-side architecture:

- input can trigger many output requests,
- all writes are funneled through one queue,
- all queue and write operations are serialized,
- the session keeps output storage alive until completion,
- the design remains safe when the `io_context` is later run on multiple threads.

5.7 Windows build guidance

5.7.1 Typical MSVC command

Assuming Boost headers are located at:

```
C:\Libraries\boost_1_89_0
```

a typical MSVC Developer Command Prompt build command is:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 writing_data_correctly.cpp
```

5.7.2 Typical MinGW-w64 command

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0
↪  writing_data_correctly.cpp -o writing_data_correctly.exe
```

5.7.3 Practical testing

A useful manual test is:

1. run the server,
2. connect with one or more TCP clients,
3. trigger multiple server responses quickly,
4. verify that the output remains ordered and complete,
5. if using multiple `io.run()` threads, verify that no corrupted output or overlapping-write failures appear.

5.8 Final safe writer implementation

5.8.1 The reusable rule

A correct session writer follows this rule:

Never start a new socket write directly from arbitrary code paths. Instead, enqueue the message into per-session owned storage, serialize queue access, and let one write-completion chain drain the queue.

5.8.2 Reusable implementation template

```
class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket)),
          strand_(socket_.get_executor())
    {
    }

    void deliver(std::string msg)
    {
        auto self = shared_from_this();

        asio::post(
            strand_,
            [self, msg = std::move(msg)]() mutable
            {
                bool write_in_progress = !self->write_queue_.empty();
                self->write_queue_.push_back(std::move(msg));

                if (!write_in_progress)
                    self->do_write();
            });
```

```
    }

private:
    void do_write()
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(write_queue_.front()),
            asio::bind_executor(
                strand_,
                [self](boost::system::error_code ec, std::size_t)
                {
                    if (ec)
                        return;

                    self->write_queue_.pop_front();

                    if (!self->write_queue_.empty())
                        self->do_write();
                }));
    }

private:
    tcp::socket socket_;
    asio::strand<asio::any_io_executor> strand_;
    std::deque<std::string> write_queue_;
};
```

5.8.3 Why this is the recommended baseline

This pattern is an excellent baseline because it solves the most important write-side correctness problems at once:

- full-message sending,
- buffer lifetime,
- message stability,
- no overlapping writes,
- safe multi-threaded session serialization.

It is one of the strongest foundations for later chapters involving command routers, protocol framing, deadlines, chat-style broadcasting, HTTP responses, and coroutine-based session logic.

5.9 Key takeaways

- `async_write()` is the normal tool when the whole supplied message must be sent.
- Buffer objects do not own memory; the underlying storage must remain valid until completion.
- Never initiate overlapping writes on the same session without a deliberate serialization design.
- A per-session write queue is the standard safe pattern.
- In multi-threaded servers, serialize queue mutation and write chaining through a strand or an equivalent mechanism.

Timeouts and Stability

6.1 Introduction

A TCP server that reads and writes correctly can still behave badly if it permits dead or stalled clients to remain connected forever. In real backend systems, stability depends not only on successful I/O but also on the ability to detect inactivity, enforce deadlines, and shut down broken sessions in a controlled way.
Boost.Asio provides timer objects specifically for this purpose. In modern code, the standard timer for relative deadlines is `boost::asio::steady_timer`. It is based on a monotonic clock and is therefore suitable for durations such as:

- idle timeout,
- write timeout,
- connection handshake timeout,
- periodic health checks,
- delayed retry logic.

For session-oriented server design, the most practical timeout model is:

1. attach one timer to each session,

2. arm the timer whenever activity should be observed within a bounded time,
3. refresh or re-arm the timer when valid activity occurs,
4. when the timer fires, terminate the session cleanly.

This chapter focuses on the most reusable session-level timeout design for Windows using modern Boost.Asio.

6.2 Add timer to session

6.2.1 Why the timer belongs inside the session

The timeout policy for one client connection usually depends on that connection's own activity. Therefore the timer naturally belongs to the session object, alongside:

- the connected socket,
- the read buffer,
- the write queue,
- protocol parsing state.

This keeps the timeout logic local to the client state it protects.

6.2.2 Minimal session with a timer member

A session timer is usually stored as a member:

```
#include <array>
#include <memory>
```

```
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;

class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket)),
          timer_(socket_.get_executor())
    {
    }

private:
    tcp::socket socket_;
    asio::steady_timer timer_;
    std::array<char, 4096> data_{};
};
```

Constructing the timer from the socket's executor is a clean modern style because it associates the timer with the same execution context as the session socket.

6.2.3 Arming the timer

A timer is armed by setting its expiry and starting an asynchronous wait:

```
void start_timer()
{
    auto self = shared_from_this();
```

```
4
5       timer_.expires_after(std::chrono::seconds(30));
6
7       timer_.async_wait(
8           [self](const boost::system::error_code& ec)
9           {
10              if (ec)
11                  return;
12
13              self->on_timeout();
14          });
15  }
```

This expresses a simple rule:

> If 30 seconds pass and the timer is not cancelled or refreshed first, the timeout handler runs.

6.2.4 What the timer wait means

The timer's `async_wait()` operation is itself an asynchronous operation exactly like socket reads and writes:

- it starts immediately,
- it completes later,
- its completion handler runs only while the associated executor is being pumped,
- if cancelled, the handler receives `boost::asio::error::operation_aborted`.

That last behavior is especially important for session timeouts.

6.2.5 Refreshing the timer

A session usually needs to treat valid I/O as evidence that the client is still alive. Therefore the timeout should be refreshed whenever activity occurs.
A common helper is:

```
void refresh_timer()
{
    timer_.expires_after(std::chrono::seconds(30));
}
```

In Boost.Asio, changing a timer's expiry while an asynchronous wait is pending cancels the pending wait. The cancelled wait handler then completes with `operation_aborted`. This behavior is the foundation of the standard refresh pattern. A refresh therefore means:

1. cancel the old wait implicitly by changing the expiry,
2. start a new wait for the new deadline.

6.2.6 Correct re-arm pattern

Because a changed expiry cancels any pending wait, the timer logic is usually written as one explicit re-arm operation:

```
void arm_idle_timeout()
{
    auto self = shared_from_this();

    timer_.expires_after(std::chrono::seconds(30));

    timer_.async_wait(
```

```
        [self](const boost::system::error_code& ec)
        {
            if (ec == asio::error::operation_aborted)
                return;

            if (ec)
                return;

            self->on_timeout();
        });
}
```

This is safer and clearer than thinking of the timer as a fire-and-forget object.

6.3 Idle timeout

6.3.1 What an idle timeout means

An idle timeout is not the same as a total connection lifetime. It means:

> If no meaningful activity occurs for a specified duration, close the session.

Typical activity that resets an idle timeout may include:

- successful read of incoming data,
- successful write of protocol output,
- successful completion of an application-level heartbeat,
- authenticated protocol step.

The exact policy depends on the server.

6.3.2 Simple idle timeout model

A very common first design is:

1. when the session starts, arm a timer for 30 seconds,
2. whenever a read succeeds, re-arm the timer,
3. if the timer fires, close the socket,
4. closing the socket causes outstanding socket operations to complete with error.

This is effective because many sessions are idle precisely when they are blocked waiting for the next read.

6.3.3 Minimal idle-timeout session

The following example adds a timer to a read loop. If the client remains inactive for 30 seconds, the socket is closed.

```
#include <array>
#include <chrono>
#include <iostream>
#include <memory>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class session : public std::enable_shared_from_this<session>
{
```

```
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket)),
          timer_(socket_.get_executor())
    {
    }

    void start()
    {
        arm_idle_timeout();
        do_read();
    }

private:
    void arm_idle_timeout()
    {
        auto self = shared_from_this();

        timer_.expires_after(std::chrono::seconds(30));

        timer_.async_wait(
            [self](error_code ec)
            {
                if (ec == asio::error::operation_aborted)
                    return;

                if (ec)
                {
                    std::cerr << "timer error: "
```

```
                            << ec.message() << '\n';
                    return;
                }

                std::cout << "idle timeout reached\n";
                self->close();
            });
    }

    void do_read()
    {
        auto self = shared_from_this();

        socket_.async_read_some(
            asio::buffer(data_),
            [self](error_code ec, std::size_t length)
            {
                if (ec)
                {
                    if (ec != asio::error::eof)
                    {
                        std::cerr << "read error: "
                                  << ec.message() << '\n';
                    }
                    self->stop_timer();
                    return;
                }

                std::cout << "received " << length << " bytes\n";
```

```
                self->arm_idle_timeout();
                self->do_read();
            });
    }

    void stop_timer()
    {
        error_code ignored_ec;
        timer_.cancel(ignored_ec);
    }

    void close()
    {
        error_code ignored_ec;
        timer_.cancel(ignored_ec);
        socket_.shutdown(tcp::socket::shutdown_both, ignored_ec);
        socket_.close(ignored_ec);
    }

private:
    tcp::socket socket_;
    asio::steady_timer timer_;
    std::array<char, 4096> data_{};
};
```

6.3.4 Why this design works

This design works because:

1. each successful read re-arms the timer,
2. re-arming cancels the previous pending timer wait,
3. the cancelled timer handler observes `operation_aborted` and returns harmlessly,
4. if no read occurs in time, the timer handler closes the socket,
5. closing the socket causes any blocked socket I/O to finish with an error instead of waiting forever.

6.3.5 Why idle timeout is usually tied to reads

For many request-driven protocols, client liveness is most naturally measured by incoming activity. The server is waiting for the client to send the next command, frame, or line. Therefore a read-side timeout is often enough for a first robust design.
Later, more advanced servers may additionally apply:

- per-write timeout,
- handshake deadline,
- request processing deadline,
- heartbeat-based liveness rules.

6.4 Cancel operations safely

6.4.1 What safe cancellation means

Safe cancellation means that shutdown or timeout does not produce use-after-free errors, invalid buffer accesses, or confused protocol state. Instead, the session ends through ordinary asynchronous completion paths.

In practice, this means:

1. do not destroy objects while handlers still depend on them,
2. do not assume cancelled operations vanish silently,
3. expect completion handlers to run with cancellation-related error codes,
4. make cleanup idempotent so it can be called more than once safely.

6.4.2 Timer cancellation behavior

Boost.Asio timers are explicit about cancellation behavior:

- a pending wait can be cancelled,
- changing expiry also cancels pending waits,
- cancelled waits complete with `boost::asio::error::operation_aborted`.

Therefore, timer handlers should always check for cancellation first.

6.4.3 Safe timer handler pattern

The safe timer pattern is:

```
timer_.async_wait(
    [self](const boost::system::error_code& ec)
    {
        if (ec == asio::error::operation_aborted)
            return;

        if (ec)
```

```
            return;

        self->on_timeout();
    });
```

This prevents old cancelled waits from being mistaken for genuine timeout events.

6.4.4 Socket cancellation in practical session design

For socket operations, the practical session-level rule is simple:

> When a timeout occurs, close the socket to terminate outstanding asynchronous socket operations.

That is the most robust and portable session shutdown action for ordinary TCP server design. Once the socket is closed, pending reads and writes complete with an error, and the handlers can end naturally.

6.4.5 Why closing the socket is the correct session shutdown action

Closing the socket is powerful because it unifies shutdown:

- blocked reads stop,
- blocked writes stop,
- no further I/O can be initiated successfully,
- the session can drain to completion through already established handler paths.

This is usually more reliable than trying to manually reason about every outstanding operation individually.

6.4.6 Make close idempotent

A timeout handler, a read error handler, a write error handler, or an external shutdown path may all attempt to close the same session. Therefore the close operation should be safe to call multiple times.
A simple pattern is:

```
void close()
{
    if (stopped_)
        return;

    stopped_ = true;

    boost::system::error_code ignored_ec;
    timer_.cancel(ignored_ec);
    socket_.shutdown(tcp::socket::shutdown_both, ignored_ec);
    socket_.close(ignored_ec);
}
```

This prevents duplicate cleanup work from causing logical confusion.

6.4.7 Why a stopped flag helps

A session with timeouts can receive several near-simultaneous events:

- the timer fires,
- the socket read completes with EOF,
- a queued write completes with an error,

- an external server shutdown requests closure.

A `stopped_` flag ensures that only the first shutdown path performs real cleanup. The later ones observe that the session is already stopping or stopped.

6.4.8 Do not treat all errors as fatal logs

Once explicit timeout-based closure exists, many completion handlers will naturally observe errors during shutdown. Those are not all “unexpected failures”. For example:

- timer wait cancelled because the timer was refreshed,
- read fails because the timeout handler already closed the socket,
- write fails because the client disconnected first.

Good session code therefore logs selectively and avoids noisy shutdown traces for normal cancellation paths.

6.4.9 Session lifetime during cancellation

The `shared_from_this()` pattern remains essential during timeout handling. Even when the timeout closes the socket, the session object stays alive until all relevant handlers complete, because those handlers hold shared ownership.
This is exactly what makes timeout-driven shutdown safe.

6.5 Final robust session class

6.5.1 Design goals

A reusable robust session should provide all of the following:

1. a socket for client communication,
2. a timer for idle timeout enforcement,
3. a read loop,
4. a safe write queue,
5. serialized handler execution,
6. idempotent cleanup,
7. stable buffer ownership,
8. correct timeout refresh behavior.

6.5.2 Why serialization matters here

Once a session has both socket handlers and timer handlers, it is even more important that state changes such as:

- queue mutation,
- timeout refresh,
- timeout firing,
- session closure

occur through one consistent serialization domain. A strand is a practical way to achieve this in multi-threaded `io_context` designs.

6.5.3 Complete robust session implementation

The following class combines:

- per-session timer,
- idle timeout refresh,
- read loop,
- safe write queue,
- strand-based serialization,
- idempotent close logic.

```
#include <array>
#include <chrono>
#include <deque>
#include <iostream>
#include <memory>
#include <string>
#include <utility>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class session : public std::enable_shared_from_this<session>
{
public:
```

```
    explicit session(tcp::socket socket)
        : socket_(std::move(socket)),
          strand_(socket_.get_executor()),
          timer_(socket_.get_executor())
    {
    }

    void start()
    {
        asio::dispatch(
            strand_,
            [self = shared_from_this()]()
            {
                self->arm_idle_timeout();
                self->do_read();
            });
    }

    void deliver(std::string msg)
    {
        asio::post(
            strand_,
            [self = shared_from_this(),
             msg = std::move(msg)]() mutable
            {
                if (self->stopped_)
                    return;

                bool write_in_progress = !self->write_queue_.empty();
```

```
                self->write_queue_.push_back(std::move(msg));

                if (!write_in_progress)
                    self->do_write();
            });
    }

private:
    void arm_idle_timeout()
    {
        timer_.expires_after(std::chrono::seconds(30));

        timer_.async_wait(
            asio::bind_executor(
                strand_,
                [self = shared_from_this()](error_code ec)
                {
                    if (ec == asio::error::operation_aborted)
                        return;

                    if (ec)
                    {
                        std::cerr << "timer error: "
                                  << ec.message() << '\n';
                        return;
                    }

                    std::cout << "session idle timeout\n";
                    self->close();
```

```
            }));
    }

    void refresh_idle_timeout()
    {
        if (stopped_)
            return;

        arm_idle_timeout();
    }

    void do_read()
    {
        socket_.async_read_some(
            asio::buffer(data_),
            asio::bind_executor(
                strand_,
                [self = shared_from_this()](error_code ec, std::size_t length)
                {
                    if (self->stopped_)
                        return;

                    if (ec)
                    {
                        if (ec != asio::error::eof)
                        {
                            std::cerr << "read error: "
                                      << ec.message() << '\n';
                        }
```

```
                        self->close();
                        return;
                    }

                    self->refresh_idle_timeout();

                    std::string reply = "ECHO: ";
                    reply.append(self->data_.data(), length);
                    self->deliver(std::move(reply));

                    self->do_read();
                }));
    }

    void do_write()
    {
        asio::async_write(
            socket_,
            asio::buffer(write_queue_.front()),
            asio::bind_executor(
                strand_,
                [self = shared_from_this()](error_code ec, std::size_t)
                {
                    if (self->stopped_)
                        return;

                    if (ec)
                    {
                        std::cerr << "write error: "
```

```
                                << ec.message() << '\n';
                        self->close();
                        return;
                    }

                    self->write_queue_.pop_front();
                    self->refresh_idle_timeout();

                    if (!self->write_queue_.empty())
                        self->do_write();
                }));
    }

    void close()
    {
        if (stopped_)
            return;

        stopped_ = true;

        error_code ignored_ec;
        timer_.cancel(ignored_ec);
        socket_.shutdown(tcp::socket::shutdown_both, ignored_ec);
        socket_.close(ignored_ec);
        write_queue_.clear();
    }

private:
    tcp::socket socket_;
```

```
    asio::strand<asio::any_io_executor> strand_;
    asio::steady_timer timer_;
    std::array<char, 4096> data_{};
    std::deque<std::string> write_queue_;
    bool stopped_ = false;
};
```

6.5.4 Why this class is robust

This implementation is robust for the intended session role because:

1. timeout logic is part of the session rather than external hidden state,
2. timer, socket handlers, queue operations, and closure all execute through the same strand,
3. refreshing the timer cancels the previous wait safely,
4. cancelled timer waits are detected through `operation_aborted`,
5. timeout enforcement closes the socket, which terminates blocked I/O,
6. close logic is idempotent,
7. queued write buffers remain valid until each write completes,
8. the object lifetime remains protected by `shared_from_this()`.

6.5.5 Why timeout refresh happens after both reads and writes

Some protocols consider any successful traffic to be proof that the session is alive. In that case, it is reasonable to refresh the idle timer after:

- successful incoming read,

- successful outgoing write.

Other protocols may define idleness only in terms of missing client requests. In those systems, it may be preferable to refresh only after reads. The underlying timer architecture remains the same.

6.5.6 Extending this pattern

This robust session class is a strong base for later additions such as:

- line-based command sessions,
- binary framed protocols,
- protocol heartbeat messages,
- request-specific deadlines,
- authentication handshake timeout,
- graceful server-wide shutdown.

6.6 Windows build guidance

6.6.1 Typical MSVC compile command

Assuming Boost headers are available at:

```
C:\Libraries\boost_1_89_0
```

a typical MSVC Developer Command Prompt build is:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 timeouts_and_stability.cpp
```

6.6.2 Typical MinGW-w64 compile command

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0
↪  timeouts_and_stability.cpp -o timeouts_and_stability.exe
```

6.6.3 Practical local test

A useful local test on Windows is:

1. run the server,
2. connect with a TCP client,
3. remain idle for longer than the configured timeout,
4. verify that the session closes automatically,
5. reconnect and send traffic repeatedly,
6. verify that active traffic refreshes the timeout and keeps the session alive.

6.7 Final robust session class

6.7.1 The reusable rule

A stable session does not wait forever. It owns its timeout state, refreshes that state on valid activity, treats timer cancellation as a normal control path, and terminates blocked I/O by closing the socket through one idempotent shutdown function.

6.7.2 Reusable timeout pattern

The core reusable pattern is:

1. store one `steady_timer` in the session,
2. arm it with `expires_after()` plus `async_wait()`,
3. on activity, re-arm it,
4. in the timer handler, ignore `operation_aborted`,
5. on genuine expiry, call one centralized `close()`,
6. make `close()` safe to call multiple times.

6.8 Key takeaways

- A per-session `steady_timer` is the natural tool for idle timeout enforcement.
- Re-arming a timer cancels the previous wait, so cancelled waits must treat `operation_aborted` as normal.
- The practical way to terminate stalled socket operations is to close the socket.
- Timeout logic, write queue logic, and socket logic should share one serialization domain.
- Idempotent shutdown is essential for robust session cleanup.

Part III

HTTP Backend (Beast)

Minimal HTTP Server (Working Code)

7.1 Introduction

After building raw TCP sessions with Boost.Asio, the next major step is to move from arbitrary byte streams to a structured application protocol. For backend development, the most practical first protocol is HTTP. It is widely understood, text-based at the protocol level, easy to test with browsers and command-line tools, and already supported by a mature official Boost library: Boost.Beast.

Boost.Beast is built on top of Boost.Asio and follows the same execution philosophy:

1. create an execution context,
2. accept TCP connections,
3. read a complete HTTP request from the connected socket,
4. construct an HTTP response object,
5. write the response back to the client,
6. close or reuse the connection according to protocol rules.

This chapter intentionally begins with the smallest useful HTTP server design. The goal is not to introduce routing, file serving, middleware, or advanced keep-alive behavior yet. The goal is to show a complete working server with clear request-to-response flow and plain text output.

7.2 Why start with Beast

7.2.1 From transport bytes to message objects

In the earlier TCP chapters, the programmer had to think directly in terms of socket reads, arbitrary chunks, and protocol framing. HTTP changes that by introducing a higher-level message model. Instead of manually deciding where a request ends, the code can let Beast parse the protocol and populate strongly typed request objects.
This provides several advantages:

- request parsing is explicit and reusable,
- headers are represented by named fields,
- method, target, version, and body are exposed cleanly,
- response generation becomes structured rather than ad hoc string concatenation,
- the code remains fully integrated with Boost.Asio.

7.2.2 Why a minimal HTTP server still matters

A minimal server is not trivial. Even in its smallest useful form, it introduces several core backend concepts:

- HTTP request reading,
- response type selection,
- correct header construction,
- content-length preparation,

- message serialization back to the client.

Once these mechanics are understood, later growth into routing, JSON responses, middleware layers, REST endpoints, and coroutine-based services becomes much easier.

7.3 Full working example

7.3.1 Design of the first server

The first working server in this chapter deliberately uses a straightforward synchronous per-connection flow. That makes the HTTP mechanics extremely clear:

1. accept one TCP connection,
2. read one HTTP request,
3. generate one plain text response,
4. write the response,
5. close the connection,
6. continue accepting the next client.

This is not the highest-performance design, but it is one of the clearest possible introductions to HTTP with Beast. It is also fully aligned with the official Beast server examples that begin from simple working patterns before moving toward more advanced asynchronous designs.

7.3.2 Complete single-file server

The following program is a full working minimal HTTP server for Windows using Boost.Beast and Boost.Asio.

```cpp
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <cstdlib>
#include <iostream>
#include <memory>
#include <string>
#include <thread>
#include <vector>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

http::response<http::string_body>
make_plain_text_response(const http::request<http::string_body>& req)
{
    http::response<http::string_body> res{
        http::status::ok, req.version()
    };

    res.set(http::field::server, "Boost.Beast Minimal Server");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(false);

    res.body() =
        "Hello from a minimal Boost.Beast HTTP server on Windows.\n"
        "Method: " + std::string(req.method_string()) + "\n"
```

```
        "Target: " + std::string(req.target()) + "\n";

    res.prepare_payload();
    return res;
}

void handle_session(tcp::socket socket)
{
    try
    {
        beast::flat_buffer buffer;

        http::request<http::string_body> req;
        http::read(socket, buffer, req);

        auto res = make_plain_text_response(req);

        http::write(socket, res);

        beast::error_code ec;
        socket.shutdown(tcp::socket::shutdown_send, ec);
    }
    catch (const std::exception& ex)
    {
        std::cerr << "session error: " << ex.what() << '\n';
    }
}

int main()
```

```
{
    try
    {
        const auto address = asio::ip::make_address("0.0.0.0");
        const unsigned short port = 8080;

        asio::io_context io{1};

        tcp::acceptor acceptor(io, {address, port});

        std::cout << "HTTP server listening on http://127.0.0.1:8080\n";

        for (;;)
        {
            tcp::socket socket(io);
            acceptor.accept(socket);
            handle_session(std::move(socket));
        }
    }
    catch (const std::exception& ex)
    {
        std::cerr << "fatal error: " << ex.what() << '\n';
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

7.3.3 What this example already demonstrates

Even though the example is intentionally minimal, it already shows the essential Beast HTTP workflow:

- a connected TCP socket is accepted,
- a Beast flat buffer stores temporary read data,
- an HTTP request object receives the parsed request,
- a typed HTTP response object is created,
- `prepare_payload()` calculates headers such as content length,
- the response is written back through the same socket.

7.3.4 Windows compile guidance

Assuming Boost headers are available at:

```
C:\Libraries\boost_1_89_0
```

a typical MSVC Developer Command Prompt build looks like:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 minimal_http_server.cpp
```

A typical MinGW-w64 build on Windows looks like:

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 minimal_http_server.cpp
↪ -o minimal_http_server.exe
```

7.3.5 Simple local test

After starting the server, a local test from PowerShell may use:

```
curl http://127.0.0.1:8080/
```

Or from a browser:

```
http://127.0.0.1:8080/
```

7.4 Request → response flow

7.4.1 The end-to-end flow

A minimal Beast HTTP exchange follows a very precise sequence. Understanding this sequence is more important than memorizing the code.
The flow is:

1. the server accepts a TCP socket,
2. the server allocates a read buffer,
3. Beast reads from the socket and parses an HTTP request into a request object,
4. application code inspects the method, target, headers, and possibly body,
5. application code constructs a response object,
6. Beast serializes the response and writes it to the socket,
7. the server either closes the connection or continues for another request.

7.4.2 Reading the request

The core input step is:

```
beast::flat_buffer buffer;
http::request<http::string_body> req;

http::read(socket, buffer, req);
```

This line performs three important jobs:

- it reads bytes from the socket,
- it parses the HTTP protocol,
- it stores the result in a typed request object.

This is a major improvement over raw TCP parsing because the program no longer needs to manually search for the end of the headers or interpret the request line itself.

7.4.3 Why `flat_buffer` is used

Beast requires a dynamic buffer for parsing incoming protocol data. The most common and practical choice in introductory code is:

```
beast::flat_buffer
```

This buffer stores received bytes that Beast needs during parsing. It is not the HTTP body type itself. Instead, it is the temporary protocol input buffer that supports the parser.

7.4.4 Inspecting the request

Once the request is parsed, application logic can inspect fields such as:

- method,
- target,
- version,
- headers,
- body.

Example:

```
std::cout << "Method: " << req.method_string() << '\n';
std::cout << "Target: " << req.target() << '\n';
std::cout << "Version: " << req.version() << '\n';
```

For a first server, method and target are usually enough to generate a plain text response.

7.4.5 Constructing the response

The core response type used in this chapter is:

```
http::response<http::string_body>
```

This is a full HTTP response whose body is stored as a standard string. It is one of the most convenient types for small text responses.
Typical response construction looks like this:

```
http::response<http::string_body> res{
    http::status::ok, req.version()
};

res.set(http::field::server, "Boost.Beast Minimal Server");
res.set(http::field::content_type, "text/plain; charset=utf-8");
res.keep_alive(false);
res.body() = "Hello HTTP\n";
res.prepare_payload();
```

7.4.6 Why the request version is reused

A common first practice is to construct the response with the same HTTP version as the request:

```
http::response<http::string_body> res{
    http::status::ok, req.version()
};
```

This keeps protocol behavior consistent and is the standard pattern in Beast examples.

7.4.7 Why `prepare_payload()` matters

For string-body responses, the body content is just ordinary application data. Before writing the message, the response should call:

```
res.prepare_payload();
```

This prepares the message for serialization, especially by updating payload-related metadata such as content length where appropriate.
In practical Beast code, this is one of the most important response-finalization steps.

7.4.8 Writing the response

The output step is:

```
http::write(socket, res);
```

This serializes the HTTP response and writes it to the connected socket. In the minimal synchronous design, this call completes before the function continues.

7.4.9 Closing the connection

The minimal server in this chapter chooses a simple policy:

- handle one request,
- send one response,
- close the connection.

That policy is represented by:

```
res.keep_alive(false);
```

and then a final socket shutdown after the write.
This keeps the first server extremely easy to reason about.

7.5 Serving plain text

7.5.1 Why plain text is the best first HTTP body

For a first HTTP server, plain text is ideal because it avoids distractions:

- no HTML templating,
- no JSON library dependency,
- no file I/O,
- no MIME complexity beyond one simple content type.

The focus remains on correct request parsing and response writing.

7.5.2 Plain text response helper

A clean way to serve plain text is to isolate response construction in a helper function:

```
http::response<http::string_body>
make_plain_text_response(const http::request<http::string_body>& req)
{
    http::response<http::string_body> res{
        http::status::ok, req.version()
    };

    res.set(http::field::server, "Boost.Beast Minimal Server");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(false);

    res.body() = "Hello from the server\n";
    res.prepare_payload();
    return res;
}
```

This style is useful because it separates protocol transport logic from application response logic.

7.5.3 Serving request details as text

A very practical diagnostic server returns information about the request itself. For example:

```
res.body() =
    "Hello from a minimal HTTP server\n"
    "Method: " + std::string(req.method_string()) + "\n"
    "Target: " + std::string(req.target()) + "\n";
```

This is helpful during early testing because it proves that parsing worked correctly and that the server is seeing the expected method and target.

7.5.4 Handling only GET in the first version

A minimal backend server may choose to allow only GET for now. That can be implemented very simply:

```
http::response<http::string_body>
make_response(const http::request<http::string_body>& req)
{
    if (req.method() != http::verb::get)
    {
        http::response<http::string_body> res{
            http::status::method_not_allowed, req.version()
        };

        res.set(http::field::server, "Boost.Beast Minimal Server");
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.keep_alive(false);
        res.body() = "Only GET is supported\n";
        res.prepare_payload();
```

```
        return res;
    }

    http::response<http::string_body> res{
        http::status::ok, req.version()
    };

    res.set(http::field::server, "Boost.Beast Minimal Server");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(false);
    res.body() = "Hello GET client\n";
    res.prepare_payload();
    return res;
}
```

This is already enough to demonstrate real HTTP status handling.

7.5.5 Adding target-specific text

The next small improvement is to vary the response by target path:

```
http::response<http::string_body>
make_response(const http::request<http::string_body>& req)
{
    http::response<http::string_body> res{
        http::status::ok, req.version()
    };

    res.set(http::field::server, "Boost.Beast Minimal Server");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
```

```
    res.keep_alive(false);

    if (req.target() == "/")
    {
        res.body() = "Welcome to the home page\n";
    }
    else if (req.target() == "/health")
    {
        res.body() = "OK\n";
    }
    else
    {
        res.result(http::status::not_found);
        res.body() = "Resource not found\n";
    }

    res.prepare_payload();
    return res;
}
```

This is the earliest form of routing and is enough for a very small internal backend utility.

7.5.6 Full plain-text variant with target handling

The following full example extends the initial server slightly while remaining minimal and fully working.

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
```

```
#include <cstdlib>
#include <iostream>
#include <string>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

http::response<http::string_body>
make_response(const http::request<http::string_body>& req)
{
    if (req.method() != http::verb::get)
    {
        http::response<http::string_body> res{
            http::status::method_not_allowed, req.version()
        };

        res.set(http::field::server, "Boost.Beast Minimal Server");
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.keep_alive(false);
        res.body() = "Only GET is supported\n";
        res.prepare_payload();
        return res;
    }

    http::response<http::string_body> res{
        http::status::ok, req.version()
    };
```

```
33
34      res.set(http::field::server, "Boost.Beast Minimal Server");
35      res.set(http::field::content_type, "text/plain; charset=utf-8");
36      res.keep_alive(false);
37
38      if (req.target() == "/")
39      {
40          res.body() = "Welcome to the minimal Beast HTTP server\n";
41      }
42      else if (req.target() == "/health")
43      {
44          res.body() = "OK\n";
45      }
46      else if (req.target() == "/about")
47      {
48          res.body() = "Plain text HTTP response from Boost.Beast\n";
49      }
50      else
51      {
52          res.result(http::status::not_found);
53          res.body() = "Not found\n";
54      }
55
56      res.prepare_payload();
57      return res;
58  }
59
60  void handle_session(tcp::socket socket)
61  {
```

```
    try
    {
        beast::flat_buffer buffer;
        http::request<http::string_body> req;

        http::read(socket, buffer, req);

        auto res = make_response(req);

        http::write(socket, res);

        beast::error_code ec;
        socket.shutdown(tcp::socket::shutdown_send, ec);
    }
    catch (const std::exception& ex)
    {
        std::cerr << "session error: " << ex.what() << '\n';
    }
}

int main()
{
    try
    {
        asio::io_context io{1};
        tcp::acceptor acceptor(
            io,
            tcp::endpoint(asio::ip::make_address("0.0.0.0"), 8080));

```

```
        std::cout << "HTTP server listening on http://127.0.0.1:8080\n";

        for (;;)
        {
            tcp::socket socket(io);
            acceptor.accept(socket);
            handle_session(std::move(socket));
        }
    }
    catch (const std::exception& ex)
    {
        std::cerr << "fatal error: " << ex.what() << '\n';
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

7.6 How this minimal server maps to Beast concepts

7.6.1 Socket acceptance

The Beast HTTP layer does not replace TCP acceptance. The server still begins with:

```
tcp::acceptor
```

and

```
tcp::socket
```

because HTTP still runs over an ordinary TCP connection.

7.6.2 Protocol parsing

The key Beast upgrade appears in:

```
http::read(socket, buffer, req);
```

That single step handles message-oriented parsing far more cleanly than hand-written TCP parsing.

7.6.3 Typed response generation

Instead of writing raw protocol lines manually, the code uses:

```
http::response<http::string_body>
```

This gives a strongly typed message object with structured headers and body management.

7.6.4 Protocol serialization

Instead of hand-formatting status lines and header text, the code relies on:

```
http::write(socket, res);
```

This keeps the code correct and aligned with Beast's intended usage.

7.7 Practical notes for Windows development

7.7.1 Port selection

Port 8080 is a convenient development port on Windows because it usually avoids administrator privileges required by lower privileged ports such as 80.

7.7.2 Firewall prompt

The first run may trigger a Windows Defender Firewall prompt. For local testing with:

```
127.0.0.1
```

this is usually only a local development concern.

7.7.3 Testing with PowerShell

Useful test commands include:

```
curl http://127.0.0.1:8080/
curl http://127.0.0.1:8080/health
curl http://127.0.0.1:8080/about
curl -Method Post http://127.0.0.1:8080/
```

These commands demonstrate successful responses, simple path handling, and method rejection.

7.8 What this chapter intentionally does not cover yet

7.8.1 No asynchronous session class yet

This chapter focuses on the smallest complete Beast HTTP server. It does not yet introduce:

- asynchronous HTTP session chains,
- write queues,
- connection keep-alive loops,

- file serving,
- chunked responses,
- coroutine-based server structure.

Those belong naturally to later chapters.

7.8.2 No JSON layer yet

Plain text is used deliberately to keep the first HTTP example focused on message mechanics rather than payload format concerns.

7.8.3 No persistent connections yet

The server closes after each response for clarity. Later designs may inspect keep-alive behavior and handle multiple requests per connection.

7.9 Final reusable pattern

7.9.1 The core HTTP baseline

A minimal Beast HTTP server can be understood as this reusable sequence:

1. accept a TCP socket,
2. create a `beast::flat_buffer`,
3. read into a `http::request<http::string_body>`,
4. build a `http::response<http::string_body>`,

5. set server and content-type headers,
6. assign the body,
7. call `prepare_payload()`,
8. write the response,
9. close the socket.

7.9.2 Compact reusable handler

The following compact function summarizes the most reusable first-step Beast handler:

```
void handle_session(tcp::socket socket)
{
    beast::flat_buffer buffer;
    http::request<http::string_body> req;

    http::read(socket, buffer, req);

    http::response<http::string_body> res{
        http::status::ok, req.version()
    };

    res.set(http::field::server, "Boost.Beast Minimal Server");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(false);
    res.body() = "Hello HTTP\n";
    res.prepare_payload();

```

```
    http::write(socket, res);

    beast::error_code ec;
    socket.shutdown(tcp::socket::shutdown_send, ec);
}
```

7.10 Key takeaways

- Beast adds structured HTTP parsing and serialization on top of Boost.Asio.
- A minimal HTTP server still begins with a normal TCP acceptor and socket.
- `http::read()` parses an HTTP request into a typed request object.
- `http::response<http::string_body>` is an excellent first response type.
- `prepare_payload()` is a key step before writing a response body.
- Plain text responses are the simplest correct starting point for a first Beast backend server.

Routing (Express Style)

8.1 Introduction

Once a minimal HTTP server can read a request and return a response, the next practical need is routing. In backend work, routing means deciding which application handler should process a request based on information such as:

1. the HTTP method,
2. the request path,
3. optionally query parameters,
4. optionally headers or content type.

In a small Beast-based server, routing is not a separate framework feature. It is application logic built on top of the parsed HTTP request object. This is one of the strengths of the Beast approach: the programmer receives a clean request representation and can build routing policies in ordinary C++ without hidden framework machinery.
An Express-style design usually aims for the following qualities:

- routes are declared clearly,
- methods are checked explicitly,

- path matching is easy to read,
- handler functions are reusable,
- the routing layer stays independent from socket and session management.

This chapter builds that style step by step.

8.2 Why routing belongs above Beast

8.2.1 Beast handles HTTP messages, not application routing

Boost.Beast gives the server structured HTTP request and response types. From the request side, the program can inspect:

- method,
- method string,
- target,
- version,
- headers,
- body.

From the response side, the program can set:

- status code,
- headers,
- keep-alive behavior,

- body,
- payload metadata through `prepare_payload()`.

This means that routing should be viewed as a pure application decision layer:

> Given a parsed request, choose the correct handler and construct the correct response.

8.2.2 Why this is useful

Keeping routing above Beast and above the transport/session code has several advantages:

- the same router can be reused in synchronous, asynchronous, or coroutine-based HTTP sessions,
- handlers can be tested without a live socket,
- path matching stays separate from I/O machinery,
- the design can grow gradually from a few routes into a larger backend.

8.3 Simple router map

8.3.1 The smallest useful idea

The smallest useful router is a map from a route key to a handler function. For a simple HTTP server, a route key can be represented by:

1. one HTTP method,
2. one path string.

The handler function receives the request and returns a response.
This is one of the cleanest starting points for an Express-style design in C++.

8.3.2 Response type used in this chapter

To keep the router compact and practical, the examples in this chapter use:

```
http::response<http::string_body>
```

This keeps all handlers simple and ideal for plain text endpoints.

8.3.3 Minimal route key and handler aliases

The following definitions provide a clean foundation:

```
#include <boost/beast/http.hpp>
#include <functional>
#include <map>
#include <string>
#include <utility>

namespace http = boost::beast::http;

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;

using handler_type = std::function<response_type(const request_type&)>;

using route_key = std::pair<http::verb, std::string>;
```

This design expresses the routing problem directly:

> A route is identified by a pair of `(method, path)` and resolves to a handler.

8.3.4 Minimal router container

A simple router can then be stored as:

```
std::map<route_key, handler_type> routes;
```

This is sufficient for exact-match routing and is often enough for internal backend tools, health endpoints, small REST services, and educational examples.

8.3.5 Registering routes

Routes are registered by assigning a handler to a method-path pair:

```
routes[{http::verb::get, "/"}] =
    [](const request_type& req)
    {
        response_type res{http::status::ok, req.version()};
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.body() = "home page\n";
        res.prepare_payload();
        return res;
    };

routes[{http::verb::get, "/health"}] =
    [](const request_type& req)
    {
        response_type res{http::status::ok, req.version()};
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.body() = "OK\n";
        res.prepare_payload();
        return res;
    };
```

This already feels similar to small framework routing, except it remains plain C++ and fully explicit.

8.3.6 Why this style is attractive

This style is attractive because:

- the route table is visible and centralized,
- handlers are ordinary functions or lambdas,
- the design is strongly typed,
- no framework magic hides control flow,
- future extensions remain possible.

8.4 Method dispatch

8.4.1 Why method dispatch matters

A route is not determined by the path alone. In HTTP, the same path may legitimately support different operations for different methods. For example:

- `GET /users` may list users,
- `POST /users` may create a user,
- `DELETE /users` may be disallowed,
- `GET /users/42` may fetch one user.

Therefore a correct router must include method dispatch as a first-class decision.

8.4.2 Using Beast's method information

The request object exposes the parsed method as an `http::verb`. This is ideal for routing because the method is already normalized to a known enumeration when possible.
A very direct check is:

```
if (req.method() == http::verb::get)
{
    // GET handler path
}
else if (req.method() == http::verb::post)
{
    // POST handler path
}
```

For routing tables, using `http::verb` in the key is cleaner than repeatedly comparing method strings manually.

8.4.3 Simple manual method dispatch

Before building a router map, it is useful to see method dispatch in raw form:

```
response_type handle_request(const request_type& req)
{
    if (req.method() == http::verb::get)
    {
        response_type res{http::status::ok, req.version()};
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.body() = "GET request accepted\n";
        res.prepare_payload();
        return res;
```

```
    }

    if (req.method() == http::verb::post)
    {
        response_type res{http::status::ok, req.version()};
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.body() = "POST request accepted\n";
        res.prepare_payload();
        return res;
    }

    response_type res{http::status::method_not_allowed, req.version()};
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.body() = "Method not allowed\n";
    res.prepare_payload();
    return res;
}
```

This is correct, but it does not scale well once paths are added. That is why combining method and path in one route key is a better long-term shape.

8.4.4 Method-specific registration helpers

To make route declarations cleaner, a router class can provide dedicated helpers such as:

```
void get(const std::string& path, handler_type handler);
void post(const std::string& path, handler_type handler);
```

Internally, each helper just inserts into the same method-path route table.
This is the first step toward an Express-like feel.

8.5 Path matching

8.5.1 What path matching really means

Path matching is the step where the router compares the request target with known application paths. In a minimal router, this is exact string matching on the path portion:

- `"/"` matches the home page,
- `"/health"` matches the health endpoint,
- `"/about"` matches the about endpoint.

For a first practical router, exact matching is often enough.

8.5.2 Why the request target needs care

The request target may contain more than a clean path. For example, a target may include a query string:

```
/users?id=42
```

If the application intends to route only on the path portion, it should first normalize the target by removing the query string portion before route lookup.

8.5.3 Simple path extraction helper

A practical helper is:

```
std::string extract_path(const request_type& req)
{
    std::string target = std::string(req.target());
```

```
    auto pos = target.find('?');

    if (pos != std::string::npos)
        target.erase(pos);

    return target;
}
```

This keeps the router focused on path matching while still allowing later query parsing if needed.

8.5.4 Exact path matching example

With the helper above, exact path routing becomes straightforward:

```
response_type handle_request(const request_type& req)
{
    const std::string path = extract_path(req);

    if (req.method() == http::verb::get && path == "/")
    {
        response_type res{http::status::ok, req.version()};
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.body() = "home\n";
        res.prepare_payload();
        return res;
    }

    if (req.method() == http::verb::get && path == "/health")
    {
```

```
        response_type res{http::status::ok, req.version()};
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.body() = "OK\n";
        res.prepare_payload();
        return res;
    }

    response_type res{http::status::not_found, req.version()};
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.body() = "Not found\n";
    res.prepare_payload();
    return res;
}
```

This is correct but repetitive. A route map removes that repetition cleanly.

8.5.5 Static versus dynamic paths

A first router usually handles static paths only. That means paths must match exactly as registered.
Examples of static routes:

- `"/"`,
- `"/health"`,
- `"/about"`.

Examples of dynamic routes, which are more advanced and not required for the first clean router:

- `"/users/:id"`,

- `"/orders/:order_id/items/:item_id"`.

For a minimal backend architecture, static exact matching is usually the right foundation.

8.5.6 Why exact matching is a good first step

Exact matching has important advantages:

- it is fast and predictable,
- it is easy to reason about,
- it avoids accidental ambiguity,
- it keeps the first router implementation compact,
- it can later be extended without changing the basic handler model.

8.6 Building a small reusable router

8.6.1 Design goals

A clean small router should provide:

1. route registration by method and path,
2. path normalization,
3. method-aware lookup,
4. a fallback not-found response,
5. clean handler code,
6. integration with Beast request and response types.

8.6.2 Reusable helpers for common responses

Before defining the router itself, it is helpful to define common response builders:

```
response_type make_text_response(
    const request_type& req,
    http::status status,
    std::string body)
{
    response_type res{status, req.version()};
    res.set(http::field::server, "Boost.Beast Router");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(false);
    res.body() = std::move(body);
    res.prepare_payload();
    return res;
}

response_type not_found_response(const request_type& req)
{
    return make_text_response(req, http::status::not_found, "Not found\n");
}

response_type method_not_allowed_response(const request_type& req)
{
    return make_text_response(
        req,
        http::status::method_not_allowed,
        "Method not allowed\n");
}
```

This keeps route handlers focused on application meaning instead of repeatedly rebuilding boilerplate response code.

8.6.3 Router class implementation

The following router is small, clear, and reusable:

```
#include <boost/beast/http.hpp>
#include <functional>
#include <map>
#include <string>
#include <utility>

namespace http = boost::beast::http;

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;
using handler_type  = std::function<response_type(const request_type&)>;
using route_key     = std::pair<http::verb, std::string>;

class router
{
public:
    void add(http::verb method, std::string path, handler_type handler)
    {
        routes_[{method, std::move(path)}] = std::move(handler);
    }

    void get(std::string path, handler_type handler)
    {
```

```
        add(http::verb::get, std::move(path), std::move(handler));
    }

    void post(std::string path, handler_type handler)
    {
        add(http::verb::post, std::move(path), std::move(handler));
    }

    response_type route(const request_type& req) const
    {
        const std::string path = extract_path(req);

        auto it = routes_.find({req.method(), path});
        if (it != routes_.end())
            return it->second(req);

        if (path_exists_for_other_method(path))
            return method_not_allowed_response(req);

        return not_found_response(req);
    }

private:
    static std::string extract_path(const request_type& req)
    {
        std::string target = std::string(req.target());
        auto pos = target.find('?');

        if (pos != std::string::npos)
```

```
            target.erase(pos);

        return target;
    }

    bool path_exists_for_other_method(const std::string& path) const
    {
        for (const auto& [key, handler] : routes_)
        {
            if (key.second == path)
                return true;
        }

        return false;
    }

    static response_type make_text_response(
        const request_type& req,
        http::status status,
        std::string body)
    {
        response_type res{status, req.version()};
        res.set(http::field::server, "Boost.Beast Router");
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.keep_alive(false);
        res.body() = std::move(body);
        res.prepare_payload();
        return res;
    }
```

```
82
83      static response_type not_found_response(const request_type& req)
84      {
85          return make_text_response(req, http::status::not_found, "Not found\n");
86      }
87
88      static response_type method_not_allowed_response(const request_type& req)
89      {
90          return make_text_response(
91              req,
92              http::status::method_not_allowed,
93              "Method not allowed\n");
94      }
95
96  private:
97      std::map<route_key, handler_type> routes_;
98  };
```

8.6.4 Why this router is clean

This design is clean because:

- route declaration is concise,
- route lookup logic is centralized,
- method mismatch and missing path are distinguished,
- handlers are simple and independent,
- the code remains ordinary C++ without macros or framework-specific syntax.

8.7 Clean routing example

8.7.1 Registering routes

With the router class above, route registration becomes straightforward:

```
router r;

r.get("/",
    [](const request_type& req)
    {
        return make_text_response(
            req,
            http::status::ok,
            "Welcome to the home page\n");
    });

r.get("/health",
    [](const request_type& req)
    {
        return make_text_response(
            req,
            http::status::ok,
            "OK\n");
    });

r.get("/about",
    [](const request_type& req)
    {
        return make_text_response(
```

```
            req,
            http::status::ok,
            "Minimal Express-style routing with Boost.Beast\n");
    });

r.post("/echo",
    [](const request_type& req)
    {
        return make_text_response(
            req,
            http::status::ok,
            "POST /echo received body size = " +
            std::to_string(req.body().size()) + "\n");
    });
```

To keep this self-contained, a free helper may be defined outside the router class:

```
response_type make_text_response(
    const request_type& req,
    http::status status,
    std::string body)
{
    response_type res{status, req.version()};
    res.set(http::field::server, "Boost.Beast Router");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(false);
    res.body() = std::move(body);
    res.prepare_payload();
    return res;
}
```

8.7.2 Complete clean routing example

The following complete program integrates a Beast HTTP session with the router:

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <map>
#include <string>
#include <utility>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;
using handler_type  = std::function<response_type(const request_type&)>;
using route_key     = std::pair<http::verb, std::string>;

response_type make_text_response(
    const request_type& req,
    http::status status,
    std::string body)
{
    response_type res{status, req.version()};
```

```
    res.set(http::field::server, "Boost.Beast Router");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(false);
    res.body() = std::move(body);
    res.prepare_payload();
    return res;
}

class router
{
public:
    void add(http::verb method, std::string path, handler_type handler)
    {
        routes_[{method, std::move(path)}] = std::move(handler);
    }

    void get(std::string path, handler_type handler)
    {
        add(http::verb::get, std::move(path), std::move(handler));
    }

    void post(std::string path, handler_type handler)
    {
        add(http::verb::post, std::move(path), std::move(handler));
    }

    response_type route(const request_type& req) const
    {
        const std::string path = extract_path(req);
```

```
56
57          auto it = routes_.find({req.method(), path});
58          if (it != routes_.end())
59              return it->second(req);
60
61          if (path_exists_for_other_method(path))
62              return make_text_response(
63                  req,
64                  http::status::method_not_allowed,
65                  "Method not allowed\n");
66
67          return make_text_response(
68              req,
69              http::status::not_found,
70              "Not found\n");
71      }
72
73  private:
74      static std::string extract_path(const request_type& req)
75      {
76          std::string target = std::string(req.target());
77          auto pos = target.find('?');
78
79          if (pos != std::string::npos)
80              target.erase(pos);
81
82          return target;
83      }
84
```

```
    bool path_exists_for_other_method(const std::string& path) const
    {
        for (const auto& [key, handler] : routes_)
        {
            if (key.second == path)
                return true;
        }

        return false;
    }

private:
    std::map<route_key, handler_type> routes_;
};

void handle_session(tcp::socket socket, const router& r)
{
    try
    {
        beast::flat_buffer buffer;
        request_type req;

        http::read(socket, buffer, req);

        auto res = r.route(req);

        http::write(socket, res);

        beast::error_code ec;
```

```
        socket.shutdown(tcp::socket::shutdown_send, ec);
    }
    catch (const std::exception& ex)
    {
        std::cerr << "session error: " << ex.what() << '\n';
    }
}

int main()
{
    try
    {
        router r;

        r.get("/",
            [](const request_type& req)
            {
                return make_text_response(
                    req,
                    http::status::ok,
                    "Welcome to the home page\n");
            });

        r.get("/health",
            [](const request_type& req)
            {
                return make_text_response(
                    req,
                    http::status::ok,
```

```
                "OK\n");
        });

    r.get("/about",
        [](const request_type& req)
        {
            return make_text_response(
                req,
                http::status::ok,
                "Minimal Express-style routing with Boost.Beast\n");
        });

    r.post("/echo",
        [](const request_type& req)
        {
            return make_text_response(
                req,
                http::status::ok,
                "POST body size = " +
                std::to_string(req.body().size()) + "\n");
        });

    asio::io_context io{1};
    tcp::acceptor acceptor(
        io,
        tcp::endpoint(asio::ip::make_address("0.0.0.0"), 8080));

    std::cout << "HTTP router listening on http://127.0.0.1:8080\n";

```

```
        for (;;)
        {
            tcp::socket socket(io);
            acceptor.accept(socket);
            handle_session(std::move(socket), r);
        }
    }
    catch (const std::exception& ex)
    {
        std::cerr << "fatal error: " << ex.what() << '\n';
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

8.7.3 What this example demonstrates

This example demonstrates the essential Express-style routing pattern in standard C++:

- route registration by method and path,
- request dispatch through one router object,
- exact path matching,
- GET and POST separation,
- 404 for unknown paths,
- 405-style behavior when a path exists but not for the requested method.

8.8 Request flow through the router

8.8.1 Operational sequence

For each HTTP request, the flow is:

1. Beast parses the request,
2. the router extracts the path from the target,
3. the router looks up `(method, path)`,
4. if found, the handler builds a response,
5. if the path exists under another method, the router returns method-not-allowed,
6. otherwise the router returns not-found,
7. Beast writes the response to the socket.

This keeps the routing logic deterministic and easy to debug.

8.8.2 Why distinguishing 404 and 405 is useful

A surprisingly useful quality in backend design is distinguishing:

- the path does not exist at all,
- the path exists but not for this method.

That distinction helps during testing, API development, and long-term maintenance. Even in a small router, it is worth preserving.

8.9 Practical Windows notes

8.9.1 Compile command

Assuming Boost headers are installed at:

```
C:\Libraries\boost_1_89_0
```

a typical MSVC Developer Command Prompt build is:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 routing_example.cpp
```

A typical MinGW-w64 build is:

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 routing_example.cpp -o
↪  routing_example.exe
```

8.9.2 Testing from PowerShell

A few useful tests are:

```
curl http://127.0.0.1:8080/
curl http://127.0.0.1:8080/health
curl http://127.0.0.1:8080/about
curl -Method Post http://127.0.0.1:8080/echo -Body "hello"
curl -Method Post http://127.0.0.1:8080/health
curl http://127.0.0.1:8080/unknown
```

These tests verify normal routing, method dispatch, path matching, and fallback behavior.

8.10 Extending this router later

8.10.1 Natural next steps

This clean router can later be extended with:

- dynamic path parameters,
- query parsing,
- middleware hooks,
- JSON request and response helpers,
- separate handler context objects,
- coroutine-based handlers,
- sub-routers or grouped prefixes.

The important point is that none of those extensions require changing the basic Beast request-response model.

8.11 Final reusable pattern

8.11.1 The pattern in one sentence

An Express-style router in Beast is a thin application layer that maps `(HTTP method, normalized path)` to a handler function returning a Beast HTTP response.

8.11.2 Compact reusable shape

The most reusable first-step shape is:

1. read a Beast request,
2. normalize the request target into a path,
3. look up a handler by `(req.method(), path)`,
4. return the handler response if found,
5. otherwise return 405 when the path exists under another method,
6. otherwise return 404.

8.12 Key takeaways

- Beast gives the structured request data needed for routing, but the router itself is application code.
- A route key based on `(http::verb, std::string)` is a strong minimal design.
- Method dispatch should be explicit and independent from path matching.
- Exact path matching is the best first routing strategy.
- A clean router keeps session I/O separate from endpoint dispatch logic.

JSON API Example

9.1 Introduction

After building routing and plain text responses, the next natural step in backend development is structured data exchange. In modern systems, JSON is the dominant format for API communication.
Boost.Beast provides the HTTP layer, while JSON handling is typically implemented using a dedicated library. In modern Boost-based backends, the most natural and fully integrated choice is:

```
boost::json
```

This chapter demonstrates how to build a clean JSON API on top of Boost.Beast, including:

- GET endpoints returning JSON,
- POST endpoints accepting JSON input,
- parsing JSON safely,
- constructing JSON responses,
- organizing a clean API structure.

The design follows official Boost practices:

- HTTP parsing via Beast,
- JSON parsing via Boost.JSON,
- explicit request-to-response flow,
- no hidden framework behavior.

9.2 Why Boost.JSON

9.2.1 Integration with Boost ecosystem

Boost.JSON is designed to integrate naturally with Boost.Asio and Beast:

- zero dependency outside Boost,
- modern C++ design,
- efficient parsing and serialization,
- value-based JSON representation.

9.2.2 Core JSON types

The main type used is:

```
boost::json::value
```

This represents any JSON value:

- object,

- array,
- string,
- number,
- boolean,
- null.

Objects are accessed via:

```
boost::json::object
```

9.3 Common helpers

9.3.1 JSON response builder

A reusable helper simplifies all JSON responses:

```
#include <boost/beast/http.hpp>
#include <boost/json.hpp>

namespace http = boost::beast::http;
namespace json = boost::json;

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;

response_type make_json_response(
    const request_type& req,
    http::status status,
```

```
    const json::value& body)
{
    response_type res{status, req.version()};

    res.set(http::field::server, "Boost.Beast JSON API");
    res.set(http::field::content_type, "application/json");
    res.keep_alive(false);

    res.body() = json::serialize(body);
    res.prepare_payload();

    return res;
}
```

9.3.2 Error response helper

```
response_type make_error(
    const request_type& req,
    http::status status,
    std::string message)
{
    json::object obj;
    obj["error"] = message;

    return make_json_response(req, status, obj);
}
```

9.4 GET endpoint

9.4.1 Purpose of GET endpoints

GET endpoints are used to retrieve data. They should:

- not modify server state,
- return structured JSON,
- be deterministic for the same input.

9.4.2 Example: system info endpoint

```
response_type handle_get_info(const request_type& req)
{
    json::object obj;

    obj["service"] = "Boost.Beast API";
    obj["version"] = "1.0";
    obj["status"]  = "running";

    return make_json_response(req, http::status::ok, obj);
}
```

9.4.3 Example: echo query-style endpoint

```
response_type handle_get_echo(const request_type& req)
{
    json::object obj;

```

```
    obj["method"] = std::string(req.method_string());
    obj["target"] = std::string(req.target());

    return make_json_response(req, http::status::ok, obj);
}
```

9.4.4 Why GET is simple

GET endpoints do not require body parsing. They rely only on:

- method,
- path,
- optionally query string.

This makes them the simplest API endpoints.

9.5 POST endpoint

9.5.1 Purpose of POST endpoints

POST endpoints are used to:

- submit data,
- create resources,
- trigger operations.

They usually include a request body.

9.5.2 Reading the request body

In Beast, the body is already available as:

```
req.body()
```

when using:

```
http::request<http::string_body>
```

9.5.3 Example: POST echo endpoint

```
response_type handle_post_echo(const request_type& req)
{
    json::value parsed;

    try
    {
        parsed = json::parse(req.body());
    }
    catch (...)
    {
        return make_error(
            req,
            http::status::bad_request,
            "Invalid JSON");
    }

    json::object result;
    result["received"] = parsed;

```

```
    return make_json_response(req, http::status::ok, result);
}
```

9.5.4 Example: POST create user

```
response_type handle_create_user(const request_type& req)
{
    json::value parsed;

    try
    {
        parsed = json::parse(req.body());
    }
    catch (...)
    {
        return make_error(req, http::status::bad_request, "Invalid JSON");
    }

    if (!parsed.is_object())
    {
        return make_error(req, http::status::bad_request, "Expected object");
    }

    auto& obj = parsed.as_object();

    if (!obj.contains("name"))
    {
        return make_error(req, http::status::bad_request, "Missing 'name'");
    }
```

```

    json::object response;

    response["id"]   = 1;
    response["name"] = obj["name"];
    response["status"] = "created";

    return make_json_response(req, http::status::ok, response);
}
```

9.6 JSON parsing and response

9.6.1 Parsing rules

JSON parsing must follow strict rules:

- invalid JSON must be rejected,
- expected structure must be validated,
- types must be checked before access.

9.6.2 Safe parsing pattern

```
json::value parsed;

try
{
    parsed = json::parse(req.body());
}
```

```
catch (...)
{
    return make_error(req, http::status::bad_request, "Invalid JSON");
}
```

9.6.3 Type validation

```
if (!parsed.is_object())
{
    return make_error(req, http::status::bad_request, "Expected JSON object");
}
```

9.6.4 Accessing fields safely

```
auto& obj = parsed.as_object();

if (obj.contains("name"))
{
    std::string name = obj["name"].as_string().c_str();
}
```

9.6.5 Building JSON response

```
json::object res;
res["message"] = "success";
res["value"]   = 42;
```

9.6.6 Serialization

```
json::serialize(res)
```

This converts the JSON object into a string suitable for HTTP response.

9.7 Clean API structure

9.7.1 Why structure matters

As APIs grow, structure becomes critical:

- handlers must remain readable,
- routing must stay clear,
- JSON logic must be isolated,
- response formatting must be consistent.

9.7.2 Recommended structure

A clean structure separates concerns:

1. routing layer,
2. handler functions,
3. JSON utilities,
4. HTTP session layer.

9.7.3 Router integration example

```
router r;

```

```
r.get("/api/info", handle_get_info);
r.get("/api/echo", handle_get_echo);
r.post("/api/echo", handle_post_echo);
r.post("/api/users", handle_create_user);
```

9.7.4 Full working JSON API example

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <boost/json.hpp>
#include <iostream>
#include <map>
#include <functional>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
namespace json  = boost::json;
using tcp = asio::ip::tcp;

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;
using handler_type  = std::function<response_type(const request_type&)>;

response_type make_json_response(
    const request_type& req,
    http::status status,
    const json::value& body)
```

```
{
    response_type res{status, req.version()};
    res.set(http::field::content_type, "application/json");
    res.body() = json::serialize(body);
    res.prepare_payload();
    return res;
}

response_type handle_get_info(const request_type& req)
{
    json::object obj;
    obj["service"] = "API";
    obj["status"]  = "ok";
    return make_json_response(req, http::status::ok, obj);
}

response_type handle_post_echo(const request_type& req)
{
    json::value parsed;

    try
    {
        parsed = json::parse(req.body());
    }
    catch (...)
    {
        json::object err;
        err["error"] = "invalid json";
        return make_json_response(req, http::status::bad_request, err);
```

```
    }

    json::object res;
    res["received"] = parsed;

    return make_json_response(req, http::status::ok, res);
}

int main()
{
    asio::io_context io{1};
    tcp::acceptor acceptor(
        io,
        tcp::endpoint(asio::ip::make_address("0.0.0.0"), 8080));

    std::map<std::pair<http::verb, std::string>, handler_type> routes;

    routes[{http::verb::get, "/api/info"}]  = handle_get_info;
    routes[{http::verb::post, "/api/echo"}] = handle_post_echo;

    for (;;)
    {
        tcp::socket socket(io);
        acceptor.accept(socket);

        beast::flat_buffer buffer;
        request_type req;

        http::read(socket, buffer, req);
```

```

        std::string path = std::string(req.target());

        auto it = routes.find({req.method(), path});

        response_type res;

        if (it != routes.end())
        {
            res = it->second(req);
        }
        else
        {
            json::object err;
            err["error"] = "not found";
            res = make_json_response(req, http::status::not_found, err);
        }

        http::write(socket, res);

        beast::error_code ec;
        socket.shutdown(tcp::socket::shutdown_send, ec);
    }
}
```

9.8 Windows build guidance

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 json_api.cpp
```

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 json_api.cpp -o
↪  json_api.exe
```

9.9 Key takeaways

- Boost.Beast handles HTTP, Boost.JSON handles structured data.
- GET endpoints return JSON without parsing input.
- POST endpoints require strict JSON parsing and validation.
- Always validate JSON type before accessing fields.
- Keep response creation centralized and consistent.
- Clean separation between routing, handlers, and JSON logic is essential for scalable backend design.

Part IV

Modern Async Style (Coroutines)

Converting to Coroutines

10.1 Introduction

Modern Boost.Asio provides first-class support for C++20 coroutines. This represents a major shift from traditional callback-based asynchronous programming toward a linear, structured, and more readable execution model.
In earlier chapters, asynchronous operations were expressed using callback chains:

```
socket_.async_read_some(..., handler);
```

Each step depended on a continuation handler, which often led to deeply nested logic and complex lifetime management.
With coroutines, the same logic becomes:

```
std::size_t n = co_await socket.async_read_some(...);
```

This transforms asynchronous control flow into a sequential style while preserving non-blocking behavior.
This chapter demonstrates:

- transforming callback chains into coroutine code,
- using `co_spawn` to launch coroutines,

- understanding `awaitable` as a return type,
- building a cleaner HTTP request flow using Boost.Beast and Asio coroutines.

10.2 Callback to coroutine transformation

10.2.1 Traditional callback-based flow

A typical asynchronous session in callback style looks like:

```
void do_read()
{
    socket_.async_read_some(
        asio::buffer(data_),
        [self = shared_from_this()](error_code ec, std::size_t length)
        {
            if (ec)
                return;

            self->do_write(length);
        });
}

void do_write(std::size_t length)
{
    asio::async_write(
        socket_,
        asio::buffer(data_, length),
        [self = shared_from_this()](error_code ec, std::size_t)
        {
```

```
            if (ec)
                return;

            self->do_read();
        });
}
```

This structure introduces several issues:

- logic is split across multiple functions,
- control flow is implicit,
- error handling is repeated,
- lifetime must be manually managed using `shared_from_this()`.

10.2.2 Coroutine-based equivalent

The same logic using coroutines becomes:

```
asio::awaitable<void> echo_session(tcp::socket socket)
{
    try
    {
        char data[1024];

        for (;;)
        {
            std::size_t n =
                co_await socket.async_read_some(
                    asio::buffer(data),
```

```
                    asio::use_awaitable);

            co_await asio::async_write(
                socket,
                asio::buffer(data, n),
                asio::use_awaitable);
        }
    }
    catch (...)
    {
    }
}
```

10.2.3 What changed

The transformation introduces several improvements:

- sequential logic replaces nested callbacks,
- no explicit handler lambdas are required,
- errors are handled using normal C++ exception flow,
- lifetime is implicitly managed by coroutine state.

10.3 `awaitable`

10.3.1 What `awaitable` represents

In Boost.Asio, a coroutine function returns:

```
asio::awaitable<T>
```

This represents an asynchronous computation that can:

- suspend execution,
- resume when an awaited operation completes,
- eventually produce a result of type `T`.

10.3.2 Basic usage

```
asio::awaitable<int> compute()
{
    co_return 42;
}
```

10.3.3 Awaiting asynchronous operations

Boost.Asio integrates coroutines using:

```
asio::use_awaitable
```

Example:

```
std::size_t n =
    co_await socket.async_read_some(
        asio::buffer(data),
        asio::use_awaitable);
```

10.3.4 Executor association

Inside a coroutine, the current executor is obtained using:

```
auto executor = co_await asio::this_coro::executor;
```

This is essential when creating new sockets or timers within a coroutine.

10.4 co_spawn

10.4.1 Purpose of co_spawn

Coroutines are not executed automatically. They must be launched using:

```
asio::co_spawn
```

This function schedules a coroutine on an executor.

10.4.2 Basic form

```
asio::co_spawn(
    executor,
    coroutine_function(),
    asio::detached);
```

10.4.3 Meaning of parameters

- executor: where the coroutine runs,
- coroutine: the function returning `awaitable`,
- completion token: defines how completion is handled.

10.4.4 Detached execution

Using:

```
asio::detached
```

means:

- the coroutine runs independently,
- no result is returned to the caller,
- suitable for session handling.

10.4.5 Example: spawning a session

```
asio::co_spawn(
    socket.get_executor(),
    echo_session(std::move(socket)),
    asio::detached);
```

10.5 Cleaner request flow

10.5.1 Coroutine-based HTTP session

The biggest advantage of coroutines appears when handling full HTTP request flows.

10.5.2 Callback-based HTTP complexity

Previously, HTTP sessions required:

- read handler,

- parse step,
- response creation,
- write handler,
- loop continuation.

Each step required a separate lambda or function.

10.5.3 Coroutine-based HTTP flow

With coroutines, the same logic becomes linear:

```
#include <boost/asio.hpp>
#include <boost/beast.hpp>
#include <boost/beast/http.hpp>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

asio::awaitable<void> http_session(tcp::socket socket)
{
    try
    {
        beast::flat_buffer buffer;

        for (;;)
        {
```

```
            http::request<http::string_body> req;

            co_await http::async_read(
                socket,
                buffer,
                req,
                asio::use_awaitable);

            http::response<http::string_body> res{
                http::status::ok, req.version()
            };

            res.set(http::field::content_type, "text/plain");
            res.body() = "Hello from coroutine HTTP server\n";
            res.prepare_payload();

            co_await http::async_write(
                socket,
                res,
                asio::use_awaitable);

            if (!res.keep_alive())
                break;
        }

        socket.shutdown(tcp::socket::shutdown_send);
    }
    catch (...)
    {
```

```
    }
}
```

10.5.4 Coroutine-based accept loop

```
asio::awaitable<void> listener(tcp::acceptor acceptor)
{
    for (;;)
    {
        tcp::socket socket =
            co_await acceptor.async_accept(asio::use_awaitable);

        asio::co_spawn(
            acceptor.get_executor(),
            http_session(std::move(socket)),
            asio::detached);
    }
}
```

10.5.5 Main function

```
int main()
{
    asio::io_context io{1};

    tcp::acceptor acceptor(
        io,
        tcp::endpoint(asio::ip::make_address("0.0.0.0"), 8080));

    asio::co_spawn(
```

```
        io,
        listener(std::move(acceptor)),
        asio::detached);

    io.run();
}
```

10.6 Why coroutines improve backend design

10.6.1 Linear control flow

The most important advantage is:

> The code reads in the same order as the logical execution.

10.6.2 Reduced cognitive load

Instead of tracking multiple callbacks, the programmer reads:

- read request,
- process,
- write response.

10.6.3 Error handling

Errors propagate naturally through exceptions instead of manual checks in every handler.

10.6.4 No explicit lifetime management

The coroutine state holds all variables, eliminating the need for:

```
shared_from_this()
```

in most session implementations.

10.7 Windows build guidance

10.7.1 MSVC

```
cl /EHsc /std:c++20 /W4 /await /I C:\Libraries\boost_1_89_0 coroutine_http.cpp
```

10.7.2 MinGW-w64

```
g++ -std=c++20 -Wall -Wextra -fcoroutines -I C:\Libraries\boost_1_89_0
↪  coroutine_http.cpp -o coroutine_http.exe
```

10.8 Final reusable pattern

10.8.1 Core transformation

- callback chain → sequential coroutine,
- handler lambdas → local variables,
- manual state → coroutine state,
- explicit continuation → `co_await`.

10.8.2 Minimal reusable coroutine session

```
asio::awaitable<void> session(tcp::socket socket)
{
    beast::flat_buffer buffer;

    for (;;)
    {
        http::request<http::string_body> req;

        co_await http::async_read(socket, buffer, req, asio::use_awaitable);

        http::response<http::string_body> res{
            http::status::ok, req.version()
        };

        res.body() = "OK\n";
        res.prepare_payload();

        co_await http::async_write(socket, res, asio::use_awaitable);

        if (!res.keep_alive())
            break;
    }
}
```

10.9 Key takeaways

- Coroutines transform asynchronous code into sequential readable logic.

- `awaitable` represents asynchronous coroutine results.
- `co_spawn` is required to start coroutine execution.
- `co_await` replaces callback chaining.
- Beast integrates naturally with coroutine-based Asio operations.
- Coroutine-based sessions are simpler, safer, and easier to maintain.

Coroutine-Based HTTP Server

11.1 Introduction

After learning the callback-based asynchronous model and then the coroutine primitives in the previous chapter, the next practical step is to rewrite the HTTP server completely in coroutine style. This is the point where the advantages of modern Boost.Asio become most visible.
A callback-based Beast server usually spreads one logical request cycle across multiple handlers:

1. accept a socket,
2. start an asynchronous read,
3. resume in a read handler,
4. construct a response,
5. start an asynchronous write,
6. resume in a write handler,
7. decide whether to continue or close.

A coroutine-based server expresses the same work in a direct top-to-bottom flow:

1. accept a socket with `co_await`,
2. read the HTTP request with `co_await`,
3. build the response in ordinary local code,
4. write the response with `co_await`,
5. repeat or close according to keep-alive state.

This does not change the asynchronous nature of the program. The server is still non-blocking at the I/O level. The change is in how the control flow is written and reasoned about.

11.2 Why rewrite the HTTP server this way

11.2.1 The transport model stays the same

Coroutines do not replace Boost.Asio or Boost.Beast. They use the same underlying asynchronous operations. The same core objects remain central:

- `tcp::acceptor`,
- `tcp::socket`,
- `beast::flat_buffer`,
- `http::request`,
- `http::response`,
- `http::async_read`,
- `http::async_write`.

The rewrite is therefore not a new architecture. It is a cleaner expression of the same asynchronous architecture.

11.2.2 What changes most

The main improvements are:

- request-handling logic becomes linear,
- state stays in ordinary local variables,
- error handling becomes easier to centralize,
- explicit continuation lambdas disappear,
- session code becomes easier to read and maintain.

11.3 Full rewrite using `co_await`

11.3.1 From callback chain to coroutine session

A callback-based HTTP session typically needs a class with member functions such as:

```
do_read();
on_read(...);
do_write(...);
on_write(...);
do_close();
```

That structure exists mainly because each asynchronous step needs a future continuation point. With coroutines, the continuation point is preserved automatically by the language. This allows the entire session to be written as one `awaitable<void>` function.

11.3.2 Minimal coroutine HTTP session

The following is the essential coroutine form of a Beast HTTP session:

```cpp
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <iostream>
#include <string>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

asio::awaitable<void> http_session(tcp::socket socket)
{
    try
    {
        beast::flat_buffer buffer;

        for (;;)
        {
            http::request<http::string_body> req;

            co_await http::async_read(
                socket,
                buffer,
                req,
                asio::use_awaitable);
```

```
27
28              http::response<http::string_body> res{
29                  http::status::ok,
30                  req.version()
31              };
32
33              res.set(http::field::server, "Boost.Beast Coroutine Server");
34              res.set(http::field::content_type, "text/plain; charset=utf-8");
35              res.keep_alive(req.keep_alive());
36              res.body() = "Hello from coroutine HTTP session\n";
37              res.prepare_payload();
38
39              co_await http::async_write(
40                  socket,
41                  res,
42                  asio::use_awaitable);
43
44              if (!res.keep_alive())
45                  break;
46          }
47
48          beast::error_code ec;
49          socket.shutdown(tcp::socket::shutdown_send, ec);
50      }
51      catch (const std::exception& ex)
52      {
53          std::cerr << "session error: " << ex.what() << '\n';
54      }
55  }
```

11.3.3 What this code expresses clearly

This one function already contains the entire per-connection HTTP lifecycle:

1. wait for a request,
2. read and parse it,
3. construct a response,
4. send it,
5. continue while keep-alive remains enabled,
6. shut down the socket when the session ends.

No session class is strictly required for this baseline design. That is one of the clearest demonstrations of the simplification gained from coroutines.

11.3.4 Coroutine listener with `co_spawn`

To make the server complete, a listener coroutine accepts incoming connections and launches one coroutine session per client.

```
asio::awaitable<void> listener(tcp::acceptor acceptor)
{
    for (;;)
    {
        tcp::socket socket =
            co_await acceptor.async_accept(asio::use_awaitable);

        asio::co_spawn(
```

```
            acceptor.get_executor(),
            http_session(std::move(socket)),
            asio::detached);
    }
}
```

11.3.5 Complete working server

The following full program is a clean coroutine rewrite of a minimal Beast HTTP server.

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <cstdlib>
#include <iostream>
#include <string>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

http::response<http::string_body>
make_response(const http::request<http::string_body>& req)
{
    http::response<http::string_body> res{
        http::status::ok,
        req.version()
    };

```

```
    res.set(http::field::server, "Boost.Beast Coroutine Server");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(req.keep_alive());

    if (req.target() == "/")
    {
        res.body() = "Welcome to the coroutine HTTP server\n";
    }
    else if (req.target() == "/health")
    {
        res.body() = "OK\n";
    }
    else
    {
        res.result(http::status::not_found);
        res.body() = "Not found\n";
    }

    res.prepare_payload();
    return res;
}

asio::awaitable<void> http_session(tcp::socket socket)
{
    try
    {
        beast::flat_buffer buffer;

        for (;;)
```

```
        {
            http::request<http::string_body> req;

            co_await http::async_read(
                socket,
                buffer,
                req,
                asio::use_awaitable);

            auto res = make_response(req);

            bool keep_alive = res.keep_alive();

            co_await http::async_write(
                socket,
                res,
                asio::use_awaitable);

            if (!keep_alive)
                break;
        }

        beast::error_code ec;
        socket.shutdown(tcp::socket::shutdown_send, ec);
    }
    catch (const std::exception& ex)
    {
        std::cerr << "session error: " << ex.what() << '\n';
    }
```

```
}

asio::awaitable<void> listener(tcp::acceptor acceptor)
{
    for (;;)
    {
        tcp::socket socket =
            co_await acceptor.async_accept(asio::use_awaitable);

        asio::co_spawn(
            acceptor.get_executor(),
            http_session(std::move(socket)),
            asio::detached);
    }
}

int main()
{
    try
    {
        asio::io_context io{1};

        tcp::acceptor acceptor(
            io,
            tcp::endpoint(asio::ip::make_address("0.0.0.0"), 8080));

        asio::co_spawn(
            io,
            listener(std::move(acceptor)),
```

```
            asio::detached);

        std::cout << "Coroutine HTTP server listening on http://127.0.0.1:8080\n";

        io.run();
    }
    catch (const std::exception& ex)
    {
        std::cerr << "fatal error: " << ex.what() << '\n';
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

11.4 Cleaner session logic

11.4.1 Why the session is easier to read

The coroutine session function keeps the entire request-response cycle in one place. This makes the code easier to understand because the programmer can read it in the same order as the actual logical work:

1. read request,
2. build response,
3. write response,
4. decide whether to continue.

In callback style, this same logic is distributed across multiple handlers, and the order of reading the source file does not necessarily match the order of execution.

11.4.2 Local variables become natural again

In callback-based code, state often has to be lifted into class members because it must survive across handler boundaries. For example:

- current request object,
- current response object,
- buffer,
- keep-alive decision,
- parser state.

In coroutine-based code, many of these values can remain local variables inside the coroutine function. The coroutine frame preserves them automatically across suspension points.

11.4.3 Cleaner request loop

A practical and readable Beast request loop looks like this:

```
for (;;)
{
    http::request<http::string_body> req;

    co_await http::async_read(
        socket,
        buffer,
```

```
        req,
        asio::use_awaitable);

    auto res = make_response(req);
    bool keep_alive = res.keep_alive();

    co_await http::async_write(
        socket,
        res,
        asio::use_awaitable);

    if (!keep_alive)
        break;
}
```

This is much closer to the mental model backend developers usually have when they describe server behavior informally.

11.4.4 Cleaner error handling

With `use_awaitable`, asynchronous failures are typically surfaced as exceptions unless the code explicitly chooses an error-code-based completion style. This means a session can often centralize error handling in one outer **`try`**/**`catch`** block.

```
asio::awaitable<void> http_session(tcp::socket socket)
{
    try
    {
        // asynchronous request loop
    }
```

```
    catch (const std::exception& ex)
    {
        std::cerr << "session error: " << ex.what() << '\n';
    }
}
```

This is usually much simpler than checking and propagating errors manually in every handler.

11.4.5 Cleaner routing integration

A router also integrates more naturally with coroutine sessions because request dispatch becomes an ordinary local function call:

```
auto res = router.route(req);
co_await http::async_write(socket, res, asio::use_awaitable);
```

No extra continuation structure is required just to move from parsing to routing to writing.

11.4.6 Coroutine version with simple router call

The following variant shows how the session stays compact even after adding application dispatch:

```
asio::awaitable<void> http_session(tcp::socket socket, const router& r)
{
    try
    {
        beast::flat_buffer buffer;

        for (;;)
        {
```

```
            http::request<http::string_body> req;

            co_await http::async_read(
                socket,
                buffer,
                req,
                asio::use_awaitable);

            auto res = r.route(req);
            bool keep_alive = res.keep_alive();

            co_await http::async_write(
                socket,
                res,
                asio::use_awaitable);

            if (!keep_alive)
                break;
        }

        beast::error_code ec;
        socket.shutdown(tcp::socket::shutdown_send, ec);
    }
    catch (const std::exception& ex)
    {
        std::cerr << "session error: " << ex.what() << '\n';
    }
}
```

11.5 Comparison with callback version

11.5.1 Typical callback structure

A callback-based Beast session often takes a shape like this:

```
class session : public std::enable_shared_from_this<session>
{
public:
    void run();
private:
    void do_read();
    void on_read(beast::error_code, std::size_t);
    void do_write();
    void on_write(bool keep_alive, beast::error_code, std::size_t);
    void do_close();

    tcp::socket socket_;
    beast::flat_buffer buffer_;
    http::request<http::string_body> req_;
};
```

This is a valid and official style, but its complexity arises from continuation management rather than from HTTP semantics alone.

11.5.2 Equivalent coroutine structure

The coroutine equivalent often reduces to this:

```
asio::awaitable<void> http_session(tcp::socket socket)
{
```

```
    beast::flat_buffer buffer;

    for (;;)
    {
        http::request<http::string_body> req;
        co_await http::async_read(socket, buffer, req, asio::use_awaitable);

        auto res = make_response(req);
        bool keep_alive = res.keep_alive();

        co_await http::async_write(socket, res, asio::use_awaitable);

        if (!keep_alive)
            break;
    }
}
```

The difference is substantial:

- fewer moving parts,
- no explicit session ownership boilerplate,
- no read-handler / write-handler split,
- no manual handoff of state between callbacks.

11.5.3 Control-flow comparison

Callback version:

1. start read,

2. return,
3. re-enter on read completion,
4. start write,
5. return,
6. re-enter on write completion,
7. decide whether to read again or close.

Coroutine version:

1. `co_await` read,
2. compute response,
3. `co_await` write,
4. continue or break.

The callback version and coroutine version describe the same runtime behavior, but the coroutine version matches the programmer's mental model much more directly.

11.5.4 State-management comparison

In callback style, state that crosses asynchronous boundaries often becomes member data:

- request object,
- response serializer or response object,
- buffer,

- continuation flags.

In coroutine style, many of these can remain local:

- request object per loop iteration,
- response object per loop iteration,
- keep-alive flag as an ordinary local boolean.

This often reduces accidental coupling inside the session implementation.

11.5.5 Error-handling comparison

Callback style often needs repeated checks such as:

```
if (ec == http::error::end_of_stream)
{
    do_close();
    return;
}

if (ec)
{
    fail(ec, "read");
    return;
}
```

Coroutine style can often centralize this with one outer exception boundary when using `use_awaitable`. That usually makes the code shorter and keeps the main path more visible.

11.5.6 Lifetime-management comparison

A callback-based session often requires:

```
shared_from_this()
```

because handlers may execute after the initiating function has returned.
A coroutine session stores its state in the coroutine frame, so a separate session class is not always necessary. This can remove one of the most difficult parts of first-time Asio server design.

11.5.7 What coroutines do not remove

Coroutines simplify expression, but they do not eliminate real backend concerns. The coroutine server still needs correct decisions about:

- keep-alive handling,
- timeouts,
- request size limits,
- routing,
- JSON parsing,
- shutdown behavior,
- concurrency model.

Coroutines are therefore a cleaner language-level asynchronous style, not a replacement for protocol discipline.

11.6 A more complete coroutine HTTP example

11.6.1 Version with simple method and path handling

The next example extends the baseline coroutine server with straightforward request dispatch.

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <cstdlib>
#include <iostream>
#include <string>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

http::response<http::string_body>
handle_request(const http::request<http::string_body>& req)
{
    http::response<http::string_body> res{
        http::status::ok,
        req.version()
    };

    res.set(http::field::server, "Boost.Beast Coroutine Server");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(req.keep_alive());

```

```
    if (req.method() != http::verb::get)
    {
        res.result(http::status::method_not_allowed);
        res.body() = "Only GET is supported\n";
        res.prepare_payload();
        return res;
    }

    if (req.target() == "/")
    {
        res.body() = "Welcome\n";
    }
    else if (req.target() == "/health")
    {
        res.body() = "OK\n";
    }
    else if (req.target() == "/about")
    {
        res.body() = "Coroutine-based HTTP server with Boost.Beast\n";
    }
    else
    {
        res.result(http::status::not_found);
        res.body() = "Not found\n";
    }

    res.prepare_payload();
    return res;
}
```

```

asio::awaitable<void> http_session(tcp::socket socket)
{
    try
    {
        beast::flat_buffer buffer;

        for (;;)
        {
            http::request<http::string_body> req;

            co_await http::async_read(
                socket,
                buffer,
                req,
                asio::use_awaitable);

            auto res = handle_request(req);
            bool keep_alive = res.keep_alive();

            co_await http::async_write(
                socket,
                res,
                asio::use_awaitable);

            if (!keep_alive)
                break;
        }

```

```
        beast::error_code ec;
        socket.shutdown(tcp::socket::shutdown_send, ec);
    }
    catch (const std::exception& ex)
    {
        std::cerr << "session error: " << ex.what() << '\n';
    }
}

asio::awaitable<void> listener(tcp::acceptor acceptor)
{
    for (;;)
    {
        tcp::socket socket =
            co_await acceptor.async_accept(asio::use_awaitable);

        asio::co_spawn(
            acceptor.get_executor(),
            http_session(std::move(socket)),
            asio::detached);
    }
}

int main()
{
    try
    {
        asio::io_context io{1};

```

```
        tcp::acceptor acceptor(
            io,
            tcp::endpoint(asio::ip::make_address("0.0.0.0"), 8080));

        asio::co_spawn(
            io,
            listener(std::move(acceptor)),
            asio::detached);

        std::cout << "Coroutine HTTP server listening on http://127.0.0.1:8080\n";

        io.run();
    }
    catch (const std::exception& ex)
    {
        std::cerr << "fatal error: " << ex.what() << '\n';
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

11.7 Practical Windows notes

11.7.1 Compiler mode

A coroutine-based server requires C++20 support and a compiler configuration that enables coroutines. On Windows, typical builds use modern MSVC or recent MinGW-w64 GCC/Clang toolchains with C++20 enabled. Official Boost.Asio coroutine support is based on

the C++20 coroutine language model together with `awaitable`, `use_awaitable`, and `co_spawn`. :contentReference[oaicite:1]index=1

11.7.2 Typical MSVC build

Assuming Boost headers are installed at:

```
C:\Libraries\boost_1_89_0
```

a typical MSVC Developer Command Prompt command is:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 coroutine_http_server.cpp
```

11.7.3 Typical MinGW-w64 build

A typical MinGW-w64 command is:

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 coroutine_http_server.cpp
↪ -o coroutine_http_server.exe
```

11.7.4 Local testing

A simple local test on Windows may use PowerShell:

```
curl http://127.0.0.1:8080/
curl http://127.0.0.1:8080/health
curl http://127.0.0.1:8080/about
curl http://127.0.0.1:8080/unknown
```

11.8 Final reusable coroutine pattern

11.8.1 The essential shape

A reusable coroutine-based HTTP server usually consists of:

1. one listener coroutine that repeatedly accepts sockets,
2. one session coroutine per socket,
3. one request loop per session,
4. one ordinary request-dispatch function that constructs the response.

11.8.2 Compact reusable template

```
asio::awaitable<void> session(tcp::socket socket)
{
    beast::flat_buffer buffer;

    for (;;)
    {
        http::request<http::string_body> req;

        co_await http::async_read(
            socket,
            buffer,
            req,
            asio::use_awaitable);

        auto res = make_response(req);
```

```
        bool keep_alive = res.keep_alive();

        co_await http::async_write(
            socket,
            res,
            asio::use_awaitable);

        if (!keep_alive)
            break;
    }
}

asio::awaitable<void> listener(tcp::acceptor acceptor)
{
    for (;;)
    {
        tcp::socket socket =
            co_await acceptor.async_accept(asio::use_awaitable);

        asio::co_spawn(
            acceptor.get_executor(),
            session(std::move(socket)),
            asio::detached);
    }
}
```

11.9 Key takeaways

- A coroutine-based Beast server keeps the same asynchronous transport model but expresses it in a much more linear form.
- A full HTTP session can often be written as one `awaitable<void>` function.
- `co_await http::async_read(..., use_awaitable)` and `co_await http::async_write(..., use_awaitable)` make the request-response path much clearer.
- `co_spawn` is the standard way to launch listener and per-connection session coroutines.
- Compared with the callback version, the coroutine version usually needs less explicit state plumbing, less manual lifetime machinery, and a cleaner top-to-bottom request flow.

Part V

Build Your Micro Framework

Minimal Express-like Framework

12.1 Introduction

Once a Beast-based HTTP server can read requests, write responses, and optionally use coroutines for cleaner control flow, the next natural step is to improve the application layer. In practice, backend code quickly becomes repetitive if every endpoint is written as a manual sequence of:

1. inspect method,
2. inspect path,
3. dispatch manually,
4. construct a response,
5. return the response.

A small framework solves this by introducing a more expressive API. The most recognizable style is the Express-like approach:

```
app.get("/health", handler);
app.post("/echo", handler);
```

This style does not replace Boost.Asio or Boost.Beast. Instead, it is a thin layer on top of them. The transport and HTTP protocol work remain handled by the official Boost libraries, while the framework organizes route registration, handler dispatch, and response construction in a cleaner form.

The goal of this chapter is not to build a large framework. It is to build a compact and correct micro framework with these properties:

- clear route registration,
- method-aware dispatch,
- reusable handler abstraction,
- compatibility with Beast request and response types,
- a full small working server.

12.2 Why an Express-like layer is useful

12.2.1 The problem with manual dispatch

A small Beast server often begins with code like this:

```
if (req.method() == http::verb::get && req.target() == "/")
{
    // response for /
}
else if (req.method() == http::verb::get && req.target() == "/health")
{
    // response for /health
}
```

```
else if (req.method() == http::verb::post && req.target() == "/echo")
{
    // response for /echo
}
else
{
    // 404 or 405
}
```

This works correctly for very small programs, but it scales poorly. As the number of endpoints grows, the dispatch code becomes harder to read, harder to maintain, and more difficult to test independently.

12.2.2 What the micro framework should improve

A micro framework should improve exactly four things:

1. declaration of routes,
2. lookup of routes,
3. structure of handlers,
4. isolation of framework code from socket/session code.

The framework should not hide Beast. It should instead provide a more readable application-facing layer while leaving the underlying request-response model explicit.

12.3 `app.get()` and `app.post()`

12.3.1 The desired developer-facing API

An Express-like interface is attractive because it expresses the intent directly:

```
app.get("/", home_handler);
app.get("/health", health_handler);
app.post("/echo", echo_handler);
```

The route declaration itself becomes readable documentation for the service.

12.3.2 Minimal design for these functions

Internally, both `app.get()` and `app.post()` are simply convenience functions that insert route entries into a routing table. The real framework concept is not the function name. The real concept is:

> Associate one HTTP method and one path with one handler function.

12.3.3 Basic request and response aliases

To keep the framework readable, it is useful to define short aliases for Beast HTTP types:

```
#include <boost/beast/http.hpp>

namespace http = boost::beast::http;

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;
```

For a first micro framework, `http::string_body` is a practical default because it works well for plain text and JSON responses.

12.3.4 First handler abstraction

The simplest useful handler type is:

```
#include <functional>

using handler_type = std::function<response_type(const request_type&)>;
```

This means every route handler receives a parsed Beast request and returns a Beast response.

12.3.5 Why this abstraction is enough for the first framework

This is enough because it keeps handlers fully independent from transport details:

- handlers do not need direct access to sockets,
- handlers do not need to know about acceptors or listeners,
- handlers do not need to manage asynchronous I/O directly,
- handlers operate purely on request and response objects.

That separation is one of the most important architectural benefits of a framework layer.

12.3.6 Basic `app.get()` and `app.post()` declarations

At the public API level, the framework should expose something like:

```
class app
{
public:
    void get(std::string path, handler_type handler);
    void post(std::string path, handler_type handler);
};
```

These functions are not complicated. Their value lies in readability and consistency.

12.4 Route registration

12.4.1 Method-path registration model

Every route in the framework is identified by:

1. an HTTP method,
2. a path string.

A very clean internal representation is:

```
using route_key = std::pair<http::verb, std::string>;
```

The routing table can then be stored as:

```
std::map<route_key, handler_type> routes_;
```

This is sufficient for a compact static-route framework.

12.4.2 Implementing registration

The core registration function is usually:

```
void add(http::verb method, std::string path, handler_type handler)
{
    routes_[{method, std::move(path)}] = std::move(handler);
}
```

Then `get()` and `post()` become thin wrappers:

```
void get(std::string path, handler_type handler)
{
    add(http::verb::get, std::move(path), std::move(handler));
}

void post(std::string path, handler_type handler)
{
    add(http::verb::post, std::move(path), std::move(handler));
}
```

12.4.3 Why this structure is strong

This route-registration model is strong because:

- it is simple,
- it is method-aware,
- it keeps exact-path dispatch efficient and predictable,
- it avoids hidden behavior,
- it is easy to extend later.

12.4.4 Path normalization

Before looking up a route, the framework should normalize the target into a path. For a first framework, the most useful normalization step is removing the query string from the target. A practical helper is:

```
static std::string extract_path(const request_type& req)
{
```

```
    std::string target = std::string(req.target());
    auto pos = target.find('?');

    if (pos != std::string::npos)
        target.erase(pos);

    return target;
}
```

This ensures route lookup is based on the actual path instead of the full target including query parameters.

12.4.5 Dispatch result categories

A correct framework should distinguish at least these cases:

1. exact route match found,
2. path exists but for a different method,
3. path does not exist at all.

This allows the framework to return appropriate responses such as:

- success,
- `405 Method Not Allowed`,
- `404 Not Found`.

12.5 Handler abstraction

12.5.1 What a handler should represent

A route handler represents application logic for one endpoint. In the micro framework, it should answer one question:

> Given this parsed request, what response should be sent back

For the first framework, that means a handler is simply a function from request to response.

12.5.2 Benefits of returning a response directly

Returning the response directly is often the cleanest first design because:

- the framework remains small,
- handlers are ordinary functions or lambdas,
- no extra response builder state is required,
- the framework can write the returned Beast response immediately.

12.5.3 Simple text-response helper

To reduce repetition, a small framework should provide a reusable helper for constructing plain text responses:

```
response_type make_text_response(
    const request_type& req,
    http::status status,
    std::string body)
```

```
{
    response_type res{status, req.version()};
    res.set(http::field::server, "Micro Framework");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(req.keep_alive());
    res.body() = std::move(body);
    res.prepare_payload();
    return res;
}
```

This helper makes route handlers far more concise.

12.5.4 Example handlers

The resulting handler functions are easy to write:

```
response_type home_handler(const request_type& req)
{
    return make_text_response(
        req,
        http::status::ok,
        "Welcome to the home page\n");
}

response_type health_handler(const request_type& req)
{
    return make_text_response(
        req,
        http::status::ok,
        "OK\n");
}
```

12.5.5 Lambda handlers

Handlers may also be lambdas, which is often convenient for small services:

```
app.get("/about",
    [](const request_type& req)
    {
        return make_text_response(
            req,
            http::status::ok,
            "Minimal Express-like framework built on Beast\n");
    });
```

12.5.6 Why this abstraction is intentionally minimal

The handler abstraction is intentionally minimal because the goal is a micro framework, not a full web platform. The framework should avoid introducing unnecessary types before they are truly needed.
Later, the same design can evolve to support:

- handler context objects,
- middleware,
- JSON helpers,
- path parameters,
- coroutine handlers.

12.6 Building the framework core

12.6.1 Core requirements

The framework core should provide:

1. route storage,
2. registration functions,
3. request dispatch,
4. fallback responses,
5. server start function.

12.6.2 Framework class structure

A compact class shape is:

```
class app
{
public:
    void get(std::string path, handler_type handler);
    void post(std::string path, handler_type handler);

    response_type handle_request(const request_type& req) const;

    void listen(const std::string& address, unsigned short port);

private:
    void add(http::verb method, std::string path, handler_type handler);
```

```
    static std::string extract_path(const request_type& req);
    bool path_exists_for_other_method(const std::string& path) const;

private:
    std::map<route_key, handler_type> routes_;
};
```

This is already enough to produce a useful small framework.

12.6.3 Dispatch implementation

The dispatch function typically follows this logic:

1. normalize the path,
2. look up `(req.method(), path)`,
3. if found, call the handler,
4. otherwise, check whether the path exists for another method,
5. return 405 or 404 accordingly.

The implementation looks like this:

```
response_type handle_request(const request_type& req) const
{
    const std::string path = extract_path(req);

    auto it = routes_.find({req.method(), path});
    if (it != routes_.end())
        return it->second(req);
```

```

    if (path_exists_for_other_method(path))
    {
        return make_text_response(
            req,
            http::status::method_not_allowed,
            "Method not allowed\n");
    }

    return make_text_response(
        req,
        http::status::not_found,
        "Not found\n");
}
```

12.6.4 Method-mismatch helper

A helper for distinguishing 404 from 405 is:

```
bool path_exists_for_other_method(const std::string& path) const
{
    for (const auto& [key, handler] : routes_)
    {
        if (key.second == path)
            return true;
    }

    return false;
}
```

This is sufficient for a compact exact-match router.

12.7 Full small framework

12.7.1 Complete implementation

The following full program provides:

- `app.get()`,
- `app.post()`,
- route registration,
- handler abstraction,
- Beast-based HTTP serving,
- a complete small working framework.

```cpp
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <map>
#include <string>
#include <utility>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;
```

```
15
16 using request_type  = http::request<http::string_body>;
17 using response_type = http::response<http::string_body>;
18 using handler_type  = std::function<response_type(const request_type&)>;
19 using route_key     = std::pair<http::verb, std::string>;
20
21 response_type make_text_response(
22     const request_type& req,
23     http::status status,
24     std::string body)
25 {
26     response_type res{status, req.version()};
27     res.set(http::field::server, "Micro Framework");
28     res.set(http::field::content_type, "text/plain; charset=utf-8");
29     res.keep_alive(req.keep_alive());
30     res.body() = std::move(body);
31     res.prepare_payload();
32     return res;
33 }
34
35 class app
36 {
37 public:
38     void get(std::string path, handler_type handler)
39     {
40         add(http::verb::get, std::move(path), std::move(handler));
41     }
42
43     void post(std::string path, handler_type handler)
```

```
    {
        add(http::verb::post, std::move(path), std::move(handler));
    }

    response_type handle_request(const request_type& req) const
    {
        const std::string path = extract_path(req);

        auto it = routes_.find({req.method(), path});
        if (it != routes_.end())
            return it->second(req);

        if (path_exists_for_other_method(path))
        {
            return make_text_response(
                req,
                http::status::method_not_allowed,
                "Method not allowed\n");
        }

        return make_text_response(
            req,
            http::status::not_found,
            "Not found\n");
    }

    void listen(const std::string& address, unsigned short port)
    {
        asio::io_context io{1};
```

```

        tcp::acceptor acceptor(
            io,
            tcp::endpoint(asio::ip::make_address(address), port));

        std::cout << "Micro framework listening on http://"
                  << address << ":" << port << '\n';

        for (;;)
        {
            tcp::socket socket(io);
            acceptor.accept(socket);
            handle_session(std::move(socket));
        }
    }

private:
    void add(http::verb method, std::string path, handler_type handler)
    {
        routes_[{method, std::move(path)}] = std::move(handler);
    }

    static std::string extract_path(const request_type& req)
    {
        std::string target = std::string(req.target());
        auto pos = target.find('?');

        if (pos != std::string::npos)
            target.erase(pos);
```

```
102
103         return target;
104     }
105
106     bool path_exists_for_other_method(const std::string& path) const
107     {
108         for (const auto& [key, handler] : routes_)
109         {
110             if (key.second == path)
111                 return true;
112         }
113
114         return false;
115     }
116
117     void handle_session(tcp::socket socket) const
118     {
119         try
120         {
121             beast::flat_buffer buffer;
122
123             for (;;)
124             {
125                 request_type req;
126
127                 http::read(socket, buffer, req);
128
129                 auto res = handle_request(req);
130                 bool keep_alive = res.keep_alive();
```

```
                http::write(socket, res);

                if (!keep_alive)
                    break;
            }

            beast::error_code ec;
            socket.shutdown(tcp::socket::shutdown_send, ec);
        }
        catch (const std::exception& ex)
        {
            std::cerr << "session error: " << ex.what() << '\n';
        }
    }

private:
    std::map<route_key, handler_type> routes_;
};

int main()
{
    try
    {
        app application;

        application.get("/",
            [](const request_type& req)
            {
```

```
            return make_text_response(
                req,
                http::status::ok,
                "Welcome to the minimal Express-like framework\n");
        });

    application.get("/health",
        [](const request_type& req)
        {
            return make_text_response(
                req,
                http::status::ok,
                "OK\n");
        });

    application.get("/about",
        [](const request_type& req)
        {
            return make_text_response(
                req,
                http::status::ok,
                "Built with Boost.Asio and Boost.Beast\n");
        });

    application.post("/echo",
        [](const request_type& req)
        {
            return make_text_response(
                req,
```

```
                    http::status::ok,
                    "POST body size = " +
                    std::to_string(req.body().size()) + "\n");
            });

        application.listen("0.0.0.0", 8080);
    }
    catch (const std::exception& ex)
    {
        std::cerr << "fatal error: " << ex.what() << '\n';
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

12.7.2 What this framework already achieves

Even though this framework is intentionally small, it already provides:

- expressive route declaration,
- exact method-path routing,
- clean handler abstraction,
- proper 404 handling,
- method-aware 405 handling,
- a reusable application object.

That is enough for many small internal tools, lightweight APIs, educational projects, and as a foundation for larger custom systems.

12.8 Using the framework

12.8.1 Example registration style

The resulting developer-facing style is clean:

```
app application;

application.get("/", home_handler);
application.get("/health", health_handler);
application.post("/echo", echo_handler);
```

This is the central achievement of the chapter. The low-level Beast session details no longer dominate the endpoint declarations.

12.8.2 Example requests

With the program running on port 8080, the following routes are available:

- `GET /`,
- `GET /health`,
- `GET /about`,
- `POST /echo`.

Unknown paths return `404 Not Found`, while a known path under the wrong method returns `405 Method Not Allowed`.

12.9 Why this is still a micro framework

12.9.1 What it intentionally does not implement

This chapter's framework does not yet include:

- middleware chain,
- path parameters,
- query parsing,
- JSON helpers,
- static file serving,
- coroutine-native handlers,
- request/response wrapper classes,
- dependency injection or service containers.

That is intentional. The purpose of a micro framework is to establish the smallest strong architecture, not to mimic a full production web framework immediately.

12.9.2 Why the minimal shape matters

A minimal shape matters because it clarifies the core ideas:

- route registration is just method-path association,
- a handler is just a function from request to response,
- the framework is just organized dispatch plus transport integration.

Once these ideas are clean, further growth becomes much easier and less error-prone.

12.10 Extending the framework later

12.10.1 Natural next steps

This micro framework can naturally grow toward:

- `app.put()` and `app.del()`,
- grouped route prefixes,
- middleware before and after handlers,
- JSON request and response helpers,
- coroutine-based request handlers,
- shared application state injection,
- path-parameter extraction.

The strength of the current design is that those features can be added without discarding the core method-path-handler model.

12.11 Windows build guidance

12.11.1 Typical MSVC command

Assuming Boost headers are located at:

```
C:\Libraries\boost_1_89_0
```

a typical MSVC Developer Command Prompt build command is:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 micro_framework.cpp
```

12.11.2 Typical MinGW-w64 command

A typical MinGW-w64 build command is:

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 micro_framework.cpp -o
↪  micro_framework.exe
```

12.11.3 Practical testing on Windows

Useful local tests include:

```
curl http://127.0.0.1:8080/
curl http://127.0.0.1:8080/health
curl http://127.0.0.1:8080/about
curl -Method Post http://127.0.0.1:8080/echo -Body "hello"
curl http://127.0.0.1:8080/echo
curl http://127.0.0.1:8080/unknown
```

These tests verify route registration, method dispatch, successful responses, 405 handling, and 404 handling.

12.12 Final reusable pattern

12.12.1 The framework idea in one sentence

A minimal Express-like framework for Beast is a thin application object that stores handlers in a method-path routing table, exposes readable registration functions such as `app.get()` and `app.post()`, and delegates transport-level HTTP I/O to Boost.Beast.

12.12.2 Compact reusable core

The most reusable first-step pattern is:

1. define request and response aliases,
2. define one handler abstraction,
3. store routes as `(http::verb, path) -> handler`,
4. expose `get()` and `post()` wrappers,
5. normalize the path,
6. dispatch to the matching handler,
7. integrate the dispatch function into a Beast HTTP session loop.

12.13 Key takeaways

- `app.get()` and `app.post()` are convenience wrappers over method-path registration.
- Route registration is simplest and strongest when based on `http::verb` plus exact path strings.
- A first handler abstraction can be just `std::function<response_type(const request_type&)>`.
- The framework should separate route dispatch from Beast socket/session mechanics.
- A small exact-match framework is enough to build a practical and readable micro backend layer.

Middleware (Practical Only)

13.1 Introduction

After building a minimal Express-like routing layer, the next step in practical backend design is middleware. Middleware provides a structured way to intercept and process requests before they reach route handlers, and optionally process responses after handlers complete.
In modern backend systems, middleware is essential for:

- logging and observability,
- authentication and authorization,
- request validation,
- cross-cutting concerns such as headers or metrics.

In this chapter, middleware is implemented in a minimal, explicit, and production-relevant way without introducing unnecessary abstraction. The goal is to build a clean and correct execution pipeline on top of the routing layer developed in the previous chapter.

13.2 Middleware model

13.2.1 Core idea

Middleware is a function that:

1. receives a request,
2. optionally performs work,
3. either:
    - passes control to the next step, or
    - returns a response immediately (short-circuit).

13.2.2 Execution chain

The execution model is a chain:

```
middleware_1 -> middleware_2 -> ... -> handler
```

Each middleware decides whether to continue or terminate the chain.

13.2.3 Minimal middleware signature

A practical and flexible middleware type is:

```
using next_type = std::function<response_type()>;

using middleware_type =
    std::function<response_type(const request_type&, next_type)>;
```

This means:

- middleware receives the request,
- it receives a callable representing the next step,
- it returns a response.

13.3 Logging middleware

13.3.1 Purpose

Logging middleware records information about incoming requests and outgoing responses. It is one of the most common and essential middleware types in backend systems.

13.3.2 Basic logging middleware

```
middleware_type logging_middleware =
    [](const request_type& req, next_type next)
{
    std::cout << "[LOG] "
              << req.method_string() << " "
              << req.target() << std::endl;

    auto res = next();

    std::cout << "[LOG] response status: "
              << static_cast<unsigned>(res.result_int())
              << std::endl;

```

```
    return res;
};
```

13.3.3 What it demonstrates

This middleware demonstrates:

- pre-processing before handler execution,
- post-processing after handler execution,
- transparent forwarding of control.

13.3.4 Extended logging example

A more detailed version may include timing:

```
#include <chrono>

middleware_type timing_middleware =
    [](const request_type& req, next_type next)
{
    auto start = std::chrono::high_resolution_clock::now();

    auto res = next();

    auto end = std::chrono::high_resolution_clock::now();
    auto duration =
        std::chrono::duration_cast<std::chrono::milliseconds>(end - start);

    std::cout << "[TIME] "
```

```
              << req.target()
              << " took " << duration.count() << " ms"
              << std::endl;

    return res;
};
```

13.4 Auth middleware

13.4.1 Purpose

Authentication middleware ensures that only authorized requests reach certain endpoints. In a minimal framework, authentication can be implemented using:

- headers,
- tokens,
- simple static keys.

13.4.2 Header-based authentication example

```
middleware_type auth_middleware =
    [](const request_type& req, next_type next)
{
    auto it = req.find(http::field::authorization);

    if (it == req.end())
    {
        return make_text_response(
```

```
            req,
            http::status::unauthorized,
            "Missing Authorization header\n");
    }

    std::string token = it->value().to_string();

    if (token != "Bearer secret_token")
    {
        return make_text_response(
            req,
            http::status::forbidden,
            "Invalid token\n");
    }

    return next();
};
```

13.4.3 Short-circuit behavior

This middleware demonstrates an important concept:

- if authentication fails, it does not call `next()`,
- instead, it returns a response immediately,
- the remaining middleware and handler are skipped.

13.4.4 Why this is important

Short-circuiting is critical for:

- security,
- performance,
- correctness of request handling.

13.5 Chain execution

13.5.1 Building the middleware pipeline

To execute middleware in order, the framework must build a chain where each middleware wraps the next.
The final step in the chain is the route handler.

13.5.2 Core execution pattern

Given:

- a list of middleware,
- a final handler,

the execution must be constructed as nested calls.

13.5.3 Pipeline builder

```
response_type execute_pipeline(
    const request_type& req,
    const std::vector<middleware_type>& middleware,
    handler_type handler)
{
```

```
    next_type next = [&]()
    {
        return handler(req);
    };

    for (auto it = middleware.rbegin(); it != middleware.rend(); ++it)
    {
        auto current = *it;

        next = [&, current, next]()
        {
            return current(req, next);
        };
    }

    return next();
}
```

13.5.4 Why reverse iteration is required

Middleware must execute in registration order:

```
m1 -> m2 -> handler
```

To construct this correctly, the wrapping must happen in reverse:

- last middleware wraps handler,
- previous middleware wraps that,
- first middleware becomes outermost.

13.5.5 Execution flow example

Given:

```
logging -> auth -> handler
```

The runtime call sequence becomes:

1. logging (before),
2. auth (before),
3. handler,
4. auth (after),
5. logging (after).

13.6 Integrating middleware into the framework

13.6.1 Extending the app class

The framework must store middleware:

```
class app
{
public:
    void use(middleware_type middleware)
    {
        middleware_.push_back(std::move(middleware));
    }

```

```
private:
    std::vector<middleware_type> middleware_;
};
```

13.6.2 Dispatch with middleware

Instead of calling the handler directly, dispatch must call the pipeline:

```
response_type handle_request(const request_type& req) const
{
    const std::string path = extract_path(req);

    auto it = routes_.find({req.method(), path});
    if (it != routes_.end())
    {
        return execute_pipeline(req, middleware_, it->second);
    }

    return make_text_response(
        req,
        http::status::not_found,
        "Not found\n");
}
```

13.7 Final pipeline example

13.7.1 Complete working example

The following program integrates:

- route registration,
- logging middleware,
- authentication middleware,
- middleware pipeline execution,
- Beast-based HTTP server.

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <functional>
#include <iostream>
#include <map>
#include <string>
#include <vector>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;
using handler_type  = std::function<response_type(const request_type&)>;
using next_type     = std::function<response_type()>;
using middleware_type =
    std::function<response_type(const request_type&, next_type)>;

```

```
response_type make_text_response(
    const request_type& req,
    http::status status,
    std::string body)
{
    response_type res{status, req.version()};
    res.set(http::field::content_type, "text/plain");
    res.keep_alive(req.keep_alive());
    res.body() = std::move(body);
    res.prepare_payload();
    return res;
}

response_type execute_pipeline(
    const request_type& req,
    const std::vector<middleware_type>& middleware,
    handler_type handler)
{
    next_type next = [&]() { return handler(req); };

    for (auto it = middleware.rbegin(); it != middleware.rend(); ++it)
    {
        auto current = *it;

        next = [&, current, next]()
        {
            return current(req, next);
        };
    }
```

```
    return next();
}

class app
{
public:
    void get(std::string path, handler_type handler)
    {
        routes_[{http::verb::get, std::move(path)}] = std::move(handler);
    }

    void use(middleware_type middleware)
    {
        middleware_.push_back(std::move(middleware));
    }

    response_type handle_request(const request_type& req) const
    {
        auto it = routes_.find({req.method(), std::string(req.target())});
        if (it != routes_.end())
            return execute_pipeline(req, middleware_, it->second);

        return make_text_response(req, http::status::not_found, "Not found\n");
    }

    void run()
    {
        asio::io_context io{1};
```

```
        tcp::acceptor acceptor(io, tcp::endpoint(tcp::v4(), 8080));

        for (;;)
        {
            tcp::socket socket(io);
            acceptor.accept(socket);

            beast::flat_buffer buffer;

            request_type req;
            http::read(socket, buffer, req);

            auto res = handle_request(req);
            http::write(socket, res);

            socket.shutdown(tcp::socket::shutdown_send);
        }
    }

private:
    std::map<std::pair<http::verb, std::string>, handler_type> routes_;
    std::vector<middleware_type> middleware_;
};

int main()
{
    app application;

    application.use(logging_middleware);
```

```
    application.use(auth_middleware);

    application.get("/secure",
        [](const request_type& req)
        {
            return make_text_response(
                req,
                http::status::ok,
                "Authorized access\n");
        });

    application.run();
}
```

13.7.2 Execution behavior

For a request to `/secure`:

1. logging middleware runs,
2. authentication middleware checks headers,
3. if valid, handler executes,
4. response flows back through middleware.

If authentication fails:

1. logging middleware runs,
2. authentication middleware returns error,
3. handler is never executed.

13.8 Key takeaways

- Middleware provides structured interception of request processing.
- Logging middleware demonstrates pre/post execution behavior.
- Authentication middleware demonstrates short-circuit control flow.
- Middleware chains are constructed by wrapping functions in reverse order.
- A simple pipeline can be implemented using `std::function` without complex frameworks.
- This design scales naturally toward more advanced backend architectures.

Part VI

Practical Projects

REST API Server (Complete)

14.1 Introduction

A practical backend server becomes much more valuable when it moves beyond single demonstration endpoints and begins to expose a small but complete REST-style API. In modern C++ with Boost.Beast, this does not require a heavy framework. The HTTP layer can remain based on Beast request and response objects, while the application layer provides route dispatch, JSON parsing, validation, and a data store.
For a first complete practical project, an in-memory CRUD server is ideal because it demonstrates the most important backend concerns in one place:

1. route dispatch,
2. request parsing,
3. JSON validation,
4. resource creation,
5. resource lookup,
6. resource update,
7. resource deletion,

8. consistent response formatting.

This chapter builds a complete REST-style server around a simple `users` resource. The storage remains in memory, which keeps the code focused on HTTP and application design rather than database integration.

14.2 Why this project matters

14.2.1 From examples to a real API shape

Small HTTP examples often demonstrate only one route such as:

```
GET /health
```

That is useful for understanding Beast, but it does not yet look like a real application. A CRUD server introduces a more realistic shape:

- `GET /users`,
- `GET /users/{id}`,
- `POST /users`,
- `PUT /users/{id}`,
- `DELETE /users/{id}`.

This shape appears constantly in internal tools, admin panels, prototypes, integration services, and educational backend projects.

14.2.2 Why in-memory storage is the right first step

A database would add many concerns that are not central to this chapter:

- schema design,
- connection management,
- transaction boundaries,
- SQL or driver API,
- persistence errors unrelated to HTTP design.

In-memory storage keeps the focus where it belongs:

- correct request handling,
- clean API structure,
- JSON validation,
- consistent response construction,
- readable server architecture.

14.3 Resource model

14.3.1 The user resource

The project in this chapter uses a small `user` resource with the following fields:

- `id`,

- `name`,
- `email`.

A compact C++ model is:

```cpp
struct user
{
    int id = 0;
    std::string name;
    std::string email;
};
```

14.3.2 Why this model is suitable

This model is suitable because:

- it is simple enough for a first REST project,
- it still needs validation,
- it naturally demonstrates create, read, update, and delete,
- it maps cleanly to JSON.

14.3.3 Serializing one user to JSON

A helper for JSON conversion keeps route handlers smaller:

```cpp
#include <boost/json.hpp>

namespace json = boost::json;

```

```
json::object to_json(const user& u)
{
    json::object obj;
    obj["id"] = u.id;
    obj["name"] = u.name;
    obj["email"] = u.email;
    return obj;
}
```

14.3.4 Serializing a list of users

```
json::array to_json_array(const std::vector<user>& users)
{
    json::array arr;

    for (const auto& u : users)
        arr.push_back(to_json(u));

    return arr;
}
```

14.4 CRUD example

14.4.1 Route plan

The CRUD API in this chapter uses these routes:

1. `GET /users` — list all users
2. `GET /users/{id}` — fetch one user

3. `POST /users` — create one user
4. `PUT /users/{id}` — replace or update one user
5. `DELETE /users/{id}` — remove one user

14.4.2 API behavior

The handlers should behave as follows:

- return JSON for all successful results,
- return suitable HTTP status codes,
- reject malformed JSON with `400 Bad Request`,
- return `404 Not Found` when a user does not exist,
- keep the handler code readable and compact.

14.4.3 Response helpers

A clean REST server should centralize response creation:

```
#include <boost/beast/http.hpp>

namespace http = boost::beast::http;

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;

response_type make_json_response(
    const request_type& req,
```

```
    http::status status,
    const json::value& body)
{
    response_type res{status, req.version()};
    res.set(http::field::server, "REST API Server");
    res.set(http::field::content_type, "application/json");
    res.keep_alive(req.keep_alive());
    res.body() = json::serialize(body);
    res.prepare_payload();
    return res;
}

response_type make_error_response(
    const request_type& req,
    http::status status,
    std::string message)
{
    json::object obj;
    obj["error"] = std::move(message);
    return make_json_response(req, status, obj);
}
```

14.4.4 GET all users

The list endpoint returns an array of all stored users:

```
response_type handle_get_users(
    const request_type& req,
    const std::vector<user>& users)
{
```

```
    return make_json_response(
        req,
        http::status::ok,
        to_json_array(users));
}
```

14.4.5 GET one user

The single-resource endpoint returns one object if found:

```
response_type handle_get_user_by_id(
    const request_type& req,
    const std::vector<user>& users,
    int id)
{
    for (const auto& u : users)
    {
        if (u.id == id)
            return make_json_response(req, http::status::ok, to_json(u));
    }

    return make_error_response(
        req,
        http::status::not_found,
        "User not found");
}
```

14.4.6 POST create user

The create endpoint parses JSON, validates required fields, allocates a new identifier, stores the user, and returns the created object:

```
response_type handle_create_user(
    const request_type& req,
    std::vector<user>& users,
    int& next_id)
{
    json::value parsed;

    try
    {
        parsed = json::parse(req.body());
    }
    catch (const std::exception&)
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Invalid JSON");
    }

    if (!parsed.is_object())
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Expected JSON object");
```

```
    }

    auto& obj = parsed.as_object();

    if (!obj.contains("name") || !obj.contains("email"))
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Missing required fields");
    }

    if (!obj["name"].is_string() || !obj["email"].is_string())
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Fields 'name' and 'email' must be strings");
    }

    user u;
    u.id = next_id++;
    u.name = std::string(obj["name"].as_string().c_str());
    u.email = std::string(obj["email"].as_string().c_str());

    users.push_back(u);

    return make_json_response(
        req,
```

```
        http::status::created,
        to_json(u));
}
```

14.4.7 PUT update user

The update endpoint replaces the mutable fields of an existing user:

```
response_type handle_update_user(
    const request_type& req,
    std::vector<user>& users,
    int id)
{
    json::value parsed;

    try
    {
        parsed = json::parse(req.body());
    }
    catch (const std::exception&)
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Invalid JSON");
    }

    if (!parsed.is_object())
    {
        return make_error_response(
```

```
            req,
            http::status::bad_request,
            "Expected JSON object");
    }

    auto& obj = parsed.as_object();

    if (!obj.contains("name") || !obj.contains("email"))
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Missing required fields");
    }

    if (!obj["name"].is_string() || !obj["email"].is_string())
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Fields 'name' and 'email' must be strings");
    }

    for (auto& u : users)
    {
        if (u.id == id)
        {
            u.name = std::string(obj["name"].as_string().c_str());
            u.email = std::string(obj["email"].as_string().c_str());
```

```
            return make_json_response(
                req,
                http::status::ok,
                to_json(u));
        }
    }

    return make_error_response(
        req,
        http::status::not_found,
        "User not found");
}
```

14.4.8 DELETE user

The delete endpoint removes the resource and returns a small JSON confirmation:

```
response_type handle_delete_user(
    const request_type& req,
    std::vector<user>& users,
    int id)
{
    for (auto it = users.begin(); it != users.end(); ++it)
    {
        if (it->id == id)
        {
            users.erase(it);

            json::object obj;
```

```
            obj["deleted"] = true;
            obj["id"] = id;

            return make_json_response(
                req,
                http::status::ok,
                obj);
        }
    }

    return make_error_response(
        req,
        http::status::not_found,
        "User not found");
}
```

14.5 In-memory storage

14.5.1 Design of the store

For this project, the store can be extremely small:

```
struct user_store
{
    std::vector<user> users;
    int next_id = 1;
};
```

This is enough for a single-threaded practical demonstration.

14.5.2 Why a vector is acceptable here

A `std::vector<user>` is acceptable because:

- the project is educational and intentionally small,
- resource counts are expected to be tiny,
- iteration is simple and readable,
- the code remains easy to understand.

For a real high-scale system, a different structure or a database would likely be preferable.

14.5.3 Store initialization

A server often benefits from sample data for quick testing:

```
user_store make_initial_store()
{
    user_store store;

    store.users.push_back({store.next_id++, "Alice", "alice@example.com"});
    store.users.push_back({store.next_id++, "Bob", "bob@example.com"});

    return store;
}
```

14.5.4 Threading note

This chapter keeps the server single-threaded for clarity. That means the in-memory store can be accessed directly without mutex protection in this example. If the server later runs handlers

concurrently across threads, the store will need explicit synchronization or a different architecture.

14.6 Path handling for resource identifiers

14.6.1 Why exact string routing is not enough

Static routes such as `/users` are easy to match exactly. Resource-specific routes such as:

```
/users/3
```

need one extra step: extracting the numeric identifier from the path.

14.6.2 Simple path parser

A compact helper for `/users/{id}` is:

```
#include <optional>

std::optional<int> parse_user_id(const std::string& path)
{
    const std::string prefix = "/users/";

    if (path.rfind(prefix, 0) != 0)
        return std::nullopt;

    std::string tail = path.substr(prefix.size());

    if (tail.empty())
        return std::nullopt;
```

```
    for (char ch : tail)
    {
        if (ch < '0' || ch > '9')
            return std::nullopt;
    }

    return std::stoi(tail);
}
```

14.6.3 Why this helper is good enough here

This helper is good enough because it is:

- explicit,
- easy to verify,
- suitable for one concrete REST pattern,
- easy to replace later with a more general router.

14.7 Clean structure

14.7.1 Recommended organization

A clean small REST server should separate the following concerns:

1. data model,
2. JSON conversion,

3. response helpers,
4. storage,
5. route dispatch,
6. Beast HTTP session loop,
7. server bootstrap.

This prevents the code from collapsing into one giant request function.

14.7.2 Structure of the complete program

The complete program in this chapter follows this order:

1. includes and aliases,
2. `user` model,
3. JSON helpers,
4. response helpers,
5. store type,
6. CRUD handlers,
7. request dispatcher,
8. Beast session handler,
9. server entry point.

14.7.3 Why this structure scales

This structure scales because later changes can be made locally:

- JSON validation changes stay in handlers,
- transport changes stay in session code,
- persistence changes stay in the store layer,
- response formatting stays centralized.

14.8 REST API server (complete)

14.8.1 Full working example

```cpp
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <boost/json.hpp>
#include <cstdlib>
#include <iostream>
#include <optional>
#include <string>
#include <vector>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
namespace json  = boost::json;
using tcp = asio::ip::tcp;
```

```

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;

struct user
{
    int id = 0;
    std::string name;
    std::string email;
};

struct user_store
{
    std::vector<user> users;
    int next_id = 1;
};

json::object to_json(const user& u)
{
    json::object obj;
    obj["id"] = u.id;
    obj["name"] = u.name;
    obj["email"] = u.email;
    return obj;
}

json::array to_json_array(const std::vector<user>& users)
{
    json::array arr;
```

```
    for (const auto& u : users)
        arr.push_back(to_json(u));

    return arr;
}

response_type make_json_response(
    const request_type& req,
    http::status status,
    const json::value& body)
{
    response_type res{status, req.version()};
    res.set(http::field::server, "REST API Server");
    res.set(http::field::content_type, "application/json");
    res.keep_alive(req.keep_alive());
    res.body() = json::serialize(body);
    res.prepare_payload();
    return res;
}

response_type make_error_response(
    const request_type& req,
    http::status status,
    std::string message)
{
    json::object obj;
    obj["error"] = std::move(message);
    return make_json_response(req, status, obj);
```

```
}

std::string extract_path(const request_type& req)
{
    std::string target = std::string(req.target());
    auto pos = target.find('?');

    if (pos != std::string::npos)
        target.erase(pos);

    return target;
}

std::optional<int> parse_user_id(const std::string& path)
{
    const std::string prefix = "/users/";

    if (path.rfind(prefix, 0) != 0)
        return std::nullopt;

    std::string tail = path.substr(prefix.size());

    if (tail.empty())
        return std::nullopt;

    for (char ch : tail)
    {
        if (ch < '0' || ch > '9')
            return std::nullopt;
```

```
    }

    return std::stoi(tail);
}

response_type handle_get_users(
    const request_type& req,
    const user_store& store)
{
    return make_json_response(
        req,
        http::status::ok,
        to_json_array(store.users));
}

response_type handle_get_user_by_id(
    const request_type& req,
    const user_store& store,
    int id)
{
    for (const auto& u : store.users)
    {
        if (u.id == id)
            return make_json_response(req, http::status::ok, to_json(u));
    }

    return make_error_response(
        req,
        http::status::not_found,
```

```
        "User not found");
}

response_type handle_create_user(
    const request_type& req,
    user_store& store)
{
    json::value parsed;

    try
    {
        parsed = json::parse(req.body());
    }
    catch (const std::exception&)
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Invalid JSON");
    }

    if (!parsed.is_object())
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Expected JSON object");
    }

```

```
    auto& obj = parsed.as_object();

    if (!obj.contains("name") || !obj.contains("email"))
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Missing required fields");
    }

    if (!obj["name"].is_string() || !obj["email"].is_string())
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Fields 'name' and 'email' must be strings");
    }

    user u;
    u.id = store.next_id++;
    u.name = std::string(obj["name"].as_string().c_str());
    u.email = std::string(obj["email"].as_string().c_str());

    store.users.push_back(u);

    return make_json_response(
        req,
        http::status::created,
        to_json(u));
```

```
}

response_type handle_update_user(
    const request_type& req,
    user_store& store,
    int id)
{
    json::value parsed;

    try
    {
        parsed = json::parse(req.body());
    }
    catch (const std::exception&)
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Invalid JSON");
    }

    if (!parsed.is_object())
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Expected JSON object");
    }

```

```
    auto& obj = parsed.as_object();

    if (!obj.contains("name") || !obj.contains("email"))
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Missing required fields");
    }

    if (!obj["name"].is_string() || !obj["email"].is_string())
    {
        return make_error_response(
            req,
            http::status::bad_request,
            "Fields 'name' and 'email' must be strings");
    }

    for (auto& u : store.users)
    {
        if (u.id == id)
        {
            u.name = std::string(obj["name"].as_string().c_str());
            u.email = std::string(obj["email"].as_string().c_str());

            return make_json_response(
                req,
                http::status::ok,
                to_json(u));
```

```
        }
    }

    return make_error_response(
        req,
        http::status::not_found,
        "User not found");
}

response_type handle_delete_user(
    const request_type& req,
    user_store& store,
    int id)
{
    for (auto it = store.users.begin(); it != store.users.end(); ++it)
    {
        if (it->id == id)
        {
            store.users.erase(it);

            json::object obj;
            obj["deleted"] = true;
            obj["id"] = id;

            return make_json_response(
                req,
                http::status::ok,
                obj);
        }
```

```
    }

    return make_error_response(
        req,
        http::status::not_found,
        "User not found");
}

response_type dispatch_request(
    const request_type& req,
    user_store& store)
{
    const std::string path = extract_path(req);

    if (req.method() == http::verb::get && path == "/users")
        return handle_get_users(req, store);

    if (auto id = parse_user_id(path))
    {
        if (req.method() == http::verb::get)
            return handle_get_user_by_id(req, store, *id);

        if (req.method() == http::verb::put)
            return handle_update_user(req, store, *id);

        if (req.method() == http::verb::delete_)
            return handle_delete_user(req, store, *id);

        return make_error_response(
```

```
            req,
            http::status::method_not_allowed,
            "Method not allowed");
    }

    if (req.method() == http::verb::post && path == "/users")
        return handle_create_user(req, store);

    return make_error_response(
        req,
        http::status::not_found,
        "Not found");
}

user_store make_initial_store()
{
    user_store store;
    store.users.push_back({store.next_id++, "Alice", "alice@example.com"});
    store.users.push_back({store.next_id++, "Bob", "bob@example.com"});
    return store;
}

void handle_session(tcp::socket socket, user_store& store)
{
    try
    {
        beast::flat_buffer buffer;

        for (;;)
```

```
        {
            request_type req;
            http::read(socket, buffer, req);

            auto res = dispatch_request(req, store);
            bool keep_alive = res.keep_alive();

            http::write(socket, res);

            if (!keep_alive)
                break;
        }

        beast::error_code ec;
        socket.shutdown(tcp::socket::shutdown_send, ec);
    }
    catch (const std::exception& ex)
    {
        std::cerr << "session error: " << ex.what() << '\n';
    }
}

int main()
{
    try
    {
        asio::io_context io{1};

        tcp::acceptor acceptor(
```

```
            io,
            tcp::endpoint(asio::ip::make_address("0.0.0.0"), 8080));

        user_store store = make_initial_store();

        std::cout << "REST API server listening on http://127.0.0.1:8080\n";

        for (;;)
        {
            tcp::socket socket(io);
            acceptor.accept(socket);
            handle_session(std::move(socket), store);
        }
    }
    catch (const std::exception& ex)
    {
        std::cerr << "fatal error: " << ex.what() << '\n';
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

14.9 How the complete server works

14.9.1 Request flow

For each connected client, the server performs the following sequence:

1. accept a TCP socket,

2. persist a Beast `flat_buffer` for HTTP parsing,
3. read one HTTP request,
4. dispatch according to method and path,
5. return a JSON response,
6. write the response,
7. continue while keep-alive remains active.

14.9.2 CRUD semantics in the implementation

The server demonstrates:

- **Create**: `POST /users`
- **Read**: `GET /users` and `GET /users/{id}`
- **Update**: `PUT /users/{id}`
- **Delete**: `DELETE /users/{id}`

14.9.3 Why the design is clean

The design is clean because each concern is separated:

- model: `user`
- storage: `user_store`
- JSON conversion: `to_json`
- response formatting: `make_json_response`

- handler logic: CRUD functions
- transport: Beast session loop
- startup: `main`

14.10 Windows build guidance

14.10.1 Typical MSVC build

Assuming Boost headers are located at:

```
C:\Libraries\boost_1_89_0
```

a typical MSVC Developer Command Prompt build command is:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 rest_api_server.cpp
```

14.10.2 Typical MinGW-w64 build

A typical MinGW-w64 build command is:

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 rest_api_server.cpp -o
↪  rest_api_server.exe
```

14.11 Practical testing

14.11.1 Useful test requests

After running the server locally, useful tests include:

```
curl http://127.0.0.1:8080/users
curl http://127.0.0.1:8080/users/1
curl -Method Post http://127.0.0.1:8080/users -Body
↪  '{"name":"Carol","email":"carol@example.com"}' -ContentType "application/json"
curl -Method Put http://127.0.0.1:8080/users/1 -Body
↪  '{"name":"Alice Updated","email":"alice.updated@example.com"}' -ContentType
↪  "application/json"
curl -Method Delete http://127.0.0.1:8080/users/2
```

14.11.2 Expected behavior

These tests verify:

- JSON list responses,
- single-resource lookup,
- successful creation,
- update behavior,
- deletion behavior,
- appropriate error responses for invalid JSON or missing users.

14.12 Final reusable pattern

14.12.1 The practical lesson

A complete small REST server with Beast does not require a large framework. The essential pattern is:

1. read a Beast HTTP request,
2. normalize the path,
3. dispatch by method and route shape,
4. parse JSON when needed,
5. apply CRUD logic to a store,
6. return a consistent JSON response.

14.12.2 What this project prepares for next

This project is a strong foundation for later enhancements such as:

- middleware,
- authentication,
- persistence with SQLite or another database,
- coroutine-based sessions,
- validation utilities,
- route parameters handled through a richer router.

14.13 Key takeaways

- A CRUD server is one of the best first complete backend projects because it combines routing, JSON parsing, validation, storage, and response design.

- In-memory storage is enough to demonstrate a real REST shape without distracting persistence complexity.
- Clean structure comes from separating model, store, JSON helpers, handlers, dispatch, and transport code.
- Beast handles HTTP message I/O cleanly, while Boost.JSON provides direct parsing and serialization for API bodies.
- A compact single-threaded REST server is already capable of expressing realistic backend patterns clearly.

WebSocket Server (Beast)

15.1 Introduction

After building HTTP servers and REST APIs, the next step in modern backend systems is real-time communication. WebSocket provides a persistent, full-duplex connection between client and server over a single TCP connection.
Boost.Beast provides a complete WebSocket implementation built on top of Boost.Asio. It integrates seamlessly with the asynchronous model already used in HTTP servers.
This chapter demonstrates:

- a minimal echo server,
- a broadcast server,
- a simple chat-like system,
- clean and correct session management.

All examples follow the official Beast design:

- use of `websocket::stream`,
- persistent `flat_buffer`,

- loop-based read/write model,
- proper error handling and shutdown.

15.2 Core WebSocket concepts

15.2.1 Upgrading from HTTP

A WebSocket connection begins as an HTTP request. The server upgrades the connection:

```
ws.accept(req);
```

After this, communication switches from HTTP messages to WebSocket frames.

15.2.2 WebSocket stream

The main type is:

```
websocket::stream<tcp::socket>
```

This wraps a TCP socket and provides:

- `read()`,
- `write()`,
- `async_read()`,
- `async_write()`.

15.2.3 Persistent buffer requirement

Beast requires a persistent buffer for reading:

```
beast::flat_buffer buffer;
```

The buffer must remain valid across reads.

15.3 Echo server

15.3.1 Purpose

The echo server is the simplest WebSocket application:

- receive a message,
- send it back unchanged.

15.3.2 Synchronous echo session

```
#include <boost/asio.hpp>
#include <boost/beast.hpp>
#include <boost/beast/websocket.hpp>
#include <iostream>

namespace asio      = boost::asio;
namespace beast     = boost::beast;
namespace websocket = beast::websocket;
using tcp = asio::ip::tcp;

void do_echo_session(tcp::socket socket)
```

```
{
    try
    {
        websocket::stream<tcp::socket> ws(std::move(socket));

        ws.accept();

        beast::flat_buffer buffer;

        for (;;)
        {
            ws.read(buffer);

            ws.text(ws.got_text());
            ws.write(buffer.data());

            buffer.consume(buffer.size());
        }
    }
    catch (const std::exception& ex)
    {
        std::cerr << "echo session error: " << ex.what() << '\n';
    }
}
```

15.3.3 Server loop

```
int main()
{
```

```
    asio::io_context io{1};

    tcp::acceptor acceptor(
        io,
        tcp::endpoint(tcp::v4(), 9002));

    std::cout << "WebSocket echo server on ws://127.0.0.1:9002\n";

    for (;;)
    {
        tcp::socket socket(io);
        acceptor.accept(socket);
        do_echo_session(std::move(socket));
    }
}
```

15.3.4 Behavior

- each client is handled sequentially,
- each message is echoed back,
- connection remains open until client closes.

15.4 Broadcast example

15.4.1 Purpose

Broadcasting sends messages from one client to all connected clients. This introduces shared state across sessions.

15.4.2 Shared session container

```
#include <set>
#include <memory>

class session;

std::set<std::shared_ptr<session>> sessions;
```

15.4.3 Session class

```
class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : ws_(std::move(socket))
    {
    }

    void run()
    {
        ws_.accept();
        sessions.insert(shared_from_this());
        do_read();
    }

private:
    void do_read()
    {
        ws_.async_read(
```

```
            buffer_,
            [self = shared_from_this()](beast::error_code ec, std::size_t)
            {
                if (ec)
                    return;

                self->broadcast();
                self->buffer_.consume(self->buffer_.size());
                self->do_read();
            });
    }

    void broadcast()
    {
        for (auto& s : sessions)
        {
            s->send(buffer_.data());
        }
    }

    void send(beast::flat_buffer::const_buffers_type data)
    {
        ws_.text(ws_.got_text());

        ws_.async_write(
            data,
            [](beast::error_code, std::size_t) {});
    }

```

```
private:
    websocket::stream<tcp::socket> ws_;
    beast::flat_buffer buffer_;
};
```

15.4.4 Server loop

```
int main()
{
    asio::io_context io{1};

    tcp::acceptor acceptor(io, tcp::endpoint(tcp::v4(), 9003));

    for (;;)
    {
        tcp::socket socket(io);
        acceptor.accept(socket);

        std::make_shared<session>(std::move(socket))->run();
    }
}
```

15.4.5 Behavior

- each client joins a global session set,
- incoming messages are broadcast,
- all connected clients receive the same message.

15.5 Chat-like system

15.5.1 Purpose

A chat system builds on broadcasting but adds:

- connection lifecycle management,
- user isolation,
- cleaner message flow.

15.5.2 Shared state

```
class shared_state
{
public:
    void join(std::shared_ptr<session> s)
    {
        sessions_.insert(s);
    }

    void leave(std::shared_ptr<session> s)
    {
        sessions_.erase(s);
    }

    void broadcast(const std::string& message)
    {
        for (auto& s : sessions_)
            s->send_text(message);
```

```
    }

private:
    std::set<std::shared_ptr<session>> sessions_;
};
```

15.5.3 Session class with shared state

```
class session : public std::enable_shared_from_this<session>
{
public:
    session(tcp::socket socket, std::shared_ptr<shared_state> state)
        : ws_(std::move(socket)), state_(state)
    {
    }

    void run()
    {
        ws_.accept();
        state_->join(shared_from_this());
        do_read();
    }

    void send_text(const std::string& msg)
    {
        ws_.text(true);

        ws_.async_write(
            asio::buffer(msg),
```

```
            [](beast::error_code, std::size_t) {});
    }

private:
    void do_read()
    {
        ws_.async_read(
            buffer_,
            [self = shared_from_this()](beast::error_code ec, std::size_t)
            {
                if (ec)
                {
                    self->state_->leave(self);
                    return;
                }

                std::string msg =
                    beast::buffers_to_string(self->buffer_.data());

                self->state_->broadcast(msg);

                self->buffer_.consume(self->buffer_.size());
                self->do_read();
            });
    }

private:
    websocket::stream<tcp::socket> ws_;
    beast::flat_buffer buffer_;
```

```
    std::shared_ptr<shared_state> state_;
};
```

15.5.4 Server

```
int main()
{
    asio::io_context io{1};

    auto state = std::make_shared<shared_state>();

    tcp::acceptor acceptor(io, tcp::endpoint(tcp::v4(), 9004));

    std::cout << "WebSocket chat server on ws://127.0.0.1:9004\n";

    for (;;)
    {
        tcp::socket socket(io);
        acceptor.accept(socket);

        std::make_shared<session>(
            std::move(socket),
            state)->run();
    }
}
```

15.5.5 Behavior

- clients join shared state,

- messages are broadcast to all clients,
- disconnections are handled cleanly,
- server maintains active session list.

15.6 Clean structure

15.6.1 Separation of concerns

A well-structured WebSocket server separates:

- transport: `tcp::acceptor`,
- protocol: `websocket::stream`,
- session: per-client logic,
- shared state: global coordination,
- application logic: message handling.

15.6.2 Why this matters

This separation ensures:

- scalability,
- maintainability,
- easier debugging,
- ability to extend features.

15.7 Windows build guidance

15.7.1 MSVC

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 websocket_server.cpp
```

15.7.2 MinGW-w64

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 websocket_server.cpp -o
↪  websocket_server.exe
```

15.8 Testing

15.8.1 Using browser console

```
let ws = new WebSocket("ws://127.0.0.1:9004");

ws.onmessage = (e) => console.log("Received:", e.data);

ws.onopen = () => ws.send("Hello from client");
```

15.9 Key takeaways

- Boost.Beast provides full WebSocket support integrated with Asio.
- Echo server demonstrates minimal read/write loop.
- Broadcast server introduces shared session management.
- Chat system demonstrates scalable design with shared state.

- Persistent buffers and correct session lifetime are critical.
- Clean separation of session and shared state enables real-world systems.

Multithreaded Server (Windows)

16.1 Introduction

A single-threaded asynchronous server can already handle many concurrent connections efficiently, because the operating system notifies the program only when socket operations are ready to make progress. However, once the workload grows, one CPU core is often not enough. At that point, the natural scaling step in Boost.Asio is not to create one `io_context` per client, and not to create one thread per connection. The standard scaling pattern is much simpler:

1. keep one shared `io_context`,
2. post all asynchronous operations through that context,
3. run that same `io_context` from multiple threads.

This creates a handler execution pool. Completion handlers for accepted sockets, reads, writes, and timers may then run on any thread currently calling `io_context::run()`.
For Beast-based servers, this is the most important practical performance step after mastering the single-threaded asynchronous model.

16.2 Why multithreading in Asio works

16.2.1 One `io_context`, many worker threads

Boost.Asio explicitly supports calling `run()` from multiple threads. The threads become equivalent workers for the same event-processing engine. When asynchronous operations complete, their handlers are scheduled onto that shared execution system.
This means a server does not need to partition connections manually in order to use multiple cores. The `io_context` itself provides the scalable handler dispatch mechanism.

16.2.2 What this changes

In a single-threaded server:

- all handlers run on one thread,
- handler sequencing is naturally serialized,
- session state is easier to reason about.

In a multithreaded server:

- handlers may run on different worker threads,
- shared state can be accessed concurrently,
- session logic must serialize operations that must not overlap.

That last point is the central correctness challenge of multithreaded Asio programming.

16.2.3 Why strands matter

A strand provides strictly sequential invocation of handlers associated with that strand. In practice, this means:

> Even if several threads are running the same `io_context`, the handlers dispatched through one strand will not execute concurrently with each other.

This is one of the most important scaling tools in Asio. It allows a session to benefit from a multithreaded server without requiring explicit mutex protection around its internal read/write chain.

16.3 Multiple threads with `io_context`

16.3.1 The standard thread-pool pattern

The standard pattern is:

1. construct one `io_context`,
2. create the acceptor and listener on that context,
3. create a configurable number of worker threads,
4. on each worker thread call `io.run()`.

This gives the server a pool of threads capable of running completion handlers.

16.3.2 Minimal thread-pool skeleton

The following code shows the basic pattern without HTTP details:

```
#include <boost/asio.hpp>
#include <iostream>
#include <thread>
#include <vector>

namespace asio = boost::asio;

int main()
{
    asio::io_context io;

    auto guard = asio::make_work_guard(io);

    std::vector<std::thread> workers;
    const std::size_t thread_count = std::thread::hardware_concurrency();

    for (std::size_t i = 0; i < thread_count; ++i)
    {
        workers.emplace_back([&io]()
        {
            io.run();
        });
    }

    // Application startup would go here.

    guard.reset();

    for (auto& t : workers)
```

```
        t.join();
}
```

16.3.3 Why a work guard is useful

A work guard prevents the `io_context` from returning immediately when there is temporarily no outstanding work. This is useful when worker threads are started before the first accept or session operation is launched.
In a practical server, the listener itself often keeps enough work outstanding, but the work guard is still a useful explicit pattern during startup and shutdown design.

16.3.4 Choosing the thread count on Windows

A practical first choice is:

```
std::thread::hardware_concurrency()
```

or a nearby value. This gives one worker thread per reported hardware thread. For many server workloads, that is a reasonable starting point, though real deployment should always be guided by measurement.

16.3.5 Why not one thread per connection

A thread-per-connection model is usually the wrong direction for modern Asio servers because:

- thread creation and scheduling overhead grow badly,
- large numbers of idle connections waste resources,
- context switching becomes expensive,
- asynchronous I/O already provides the waiting mechanism efficiently.

The whole point of Asio is to let a small thread pool drive many connections.

16.4 Scaling pattern

16.4.1 The practical Beast architecture

A scalable Beast HTTP server typically has three layers:

1. a listener object that accepts new sockets,
2. a session object per connection,
3. one shared `io_context` driven by multiple threads.

The listener accepts sockets asynchronously. For each accepted socket, it creates a session. Each session performs asynchronous HTTP reads and writes. All of this runs on the same `io_context` thread pool.

16.4.2 Why the listener usually has its own strand

The accept loop itself should remain orderly. A common design is to bind the acceptor operations to a strand so that the listener's own internal state is serialized. This does not stop accepted sessions from running across the thread pool. It simply keeps the listener logic clean and safe.

16.4.3 Why sessions should use a strand

A session usually has state such as:

- input buffer,
- current request,
- current response,

- write queue,
- timeout timer.

In a multithreaded server, these elements must not be mutated concurrently by unrelated handlers. The usual solution is to bind the session's asynchronous operations to a strand associated with the socket executor.
This allows:

- the whole server to use many worker threads,
- each session to preserve internal sequential correctness,
- multiple sessions to still run in parallel across cores.

16.4.4 The scaling rule in one sentence

A very practical rule is:

> Scale across sessions with the thread pool, and serialize within one session using a strand.

That is the core multithreaded Beast pattern on Windows and other platforms.

16.5 Asynchronous HTTP session for a multithreaded server

16.5.1 Design goals

A performance-ready session should provide:

- asynchronous HTTP read,

- asynchronous HTTP write,
- keep-alive support,
- strand-based serialization,
- safe object lifetime via `shared_from_this()`.

16.5.2 Session class

The following session class is a clean multithread-ready baseline for Beast HTTP:

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <iostream>
#include <memory>
#include <string>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

class http_session : public std::enable_shared_from_this<http_session>
{
public:
    explicit http_session(tcp::socket socket)
        : stream_(std::move(socket)),
          strand_(stream_.get_executor())
    {
```

```
    }

    void run()
    {
        asio::dispatch(
            strand_,
            beast::bind_front_handler(
                &http_session::do_read,
                shared_from_this()));
    }

private:
    void do_read()
    {
        req_ = {};

        http::async_read(
            stream_,
            buffer_,
            req_,
            asio::bind_executor(
                strand_,
                beast::bind_front_handler(
                    &http_session::on_read,
                    shared_from_this())));
    }

    void on_read(beast::error_code ec, std::size_t)
    {
```

```
        if (ec == http::error::end_of_stream)
        {
            do_close();
            return;
        }

        if (ec)
        {
            std::cerr << "read error: " << ec.message() << '\n';
            return;
        }

        handle_request();
    }

    void handle_request()
    {
        http::response<http::string_body> res{
            http::status::ok,
            req_.version()
        };

        res.set(http::field::server, "Boost.Beast Multithreaded Server");
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.keep_alive(req_.keep_alive());

        if (req_.target() == "/")
        {
            res.body() = "Hello from the multithreaded Beast server\n";
```

```
        }
        else if (req_.target() == "/health")
        {
            res.body() = "OK\n";
        }
        else
        {
            res.result(http::status::not_found);
            res.body() = "Not found\n";
        }

        res.prepare_payload();

        res_ = std::make_shared<http::response<http::string_body>>(std::move(res));

        http::async_write(
            stream_,
            *res_,
            asio::bind_executor(
                strand_,
                beast::bind_front_handler(
                    &http_session::on_write,
                    shared_from_this(),
                    res_->need_eof())));
    }

    void on_write(bool close, beast::error_code ec, std::size_t)
    {
        if (ec)
```

```
        {
            std::cerr << "write error: " << ec.message() << '\n';
            return;
        }

        res_.reset();

        if (close)
        {
            do_close();
            return;
        }

        do_read();
    }

    void do_close()
    {
        beast::error_code ec;
        stream_.socket().shutdown(tcp::socket::shutdown_send, ec);
    }

private:
    beast::tcp_stream stream_;
    asio::strand<asio::any_io_executor> strand_;
    beast::flat_buffer buffer_;
    http::request<http::string_body> req_;
    std::shared_ptr<http::response<http::string_body>> res_;
};
```

16.5.3 Why this session is safe in a multithreaded server

This session is safe for the intended design because:

1. the server may have many worker threads,
2. the session itself serializes its handlers through its strand,
3. request and response state are therefore not mutated concurrently,
4. the object lifetime is protected by `shared_from_this()`.

16.5.4 Why the response is stored by shared pointer

The asynchronous write operation needs the response object to remain alive until the completion handler executes. Storing the response in:

```
std::shared_ptr<http::response<http::string_body>>
```

is a straightforward and correct solution.

16.6 Listener for the multithreaded server

16.6.1 Role of the listener

The listener owns the acceptor and repeatedly starts asynchronous accept operations. Each new socket is handed to a newly created session.

16.6.2 Listener class

```
class listener : public std::enable_shared_from_this<listener>
{
public:
    listener(
        asio::io_context& io,
        tcp::endpoint endpoint)
        : acceptor_(asio::make_strand(io))
    {
        beast::error_code ec;

        acceptor_.open(endpoint.protocol(), ec);
        if (ec)
            throw beast::system_error(ec);

        acceptor_.set_option(asio::socket_base::reuse_address(true), ec);
        if (ec)
            throw beast::system_error(ec);

        acceptor_.bind(endpoint, ec);
        if (ec)
            throw beast::system_error(ec);

        acceptor_.listen(asio::socket_base::max_listen_connections, ec);
        if (ec)
            throw beast::system_error(ec);
    }

```

```
    void run()
    {
        do_accept();
    }

private:
    void do_accept()
    {
        acceptor_.async_accept(
            asio::make_strand(acceptor_.get_executor()),
            beast::bind_front_handler(
                &listener::on_accept,
                shared_from_this()));
    }

    void on_accept(beast::error_code ec, tcp::socket socket)
    {
        if (!ec)
        {
            std::make_shared<http_session>(std::move(socket))->run();
        }
        else
        {
            std::cerr << "accept error: " << ec.message() << '\n';
        }

        do_accept();
    }

```

```
private:
    tcp::acceptor acceptor_;
};
```

16.6.3 Why accepted sockets get their own strand

The call:

```
asio::make_strand(acceptor_.get_executor())
```

ensures the accepted socket is associated with a strand-aware executor. This fits naturally with the session design, where the session serializes its own handlers.

16.7 Performance-ready version

16.7.1 Complete multithreaded server

The following full program combines:

- one `io_context`,
- one asynchronous Beast listener,
- one strand-serialized session per connection,
- a thread pool calling `io.run()`.

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <cstdlib>
#include <iostream>
```

```
#include <memory>
#include <string>
#include <thread>
#include <vector>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

class http_session : public std::enable_shared_from_this<http_session>
{
public:
    explicit http_session(tcp::socket socket)
        : stream_(std::move(socket)),
          strand_(stream_.get_executor())
    {
    }

    void run()
    {
        asio::dispatch(
            strand_,
            beast::bind_front_handler(
                &http_session::do_read,
                shared_from_this()));
    }

private:
```

```
    void do_read()
    {
        req_ = {};

        http::async_read(
            stream_,
            buffer_,
            req_,
            asio::bind_executor(
                strand_,
                beast::bind_front_handler(
                    &http_session::on_read,
                    shared_from_this())));
    }

    void on_read(beast::error_code ec, std::size_t)
    {
        if (ec == http::error::end_of_stream)
        {
            do_close();
            return;
        }

        if (ec)
        {
            std::cerr << "read error: " << ec.message() << '\n';
            return;
        }

```

```
        handle_request();
    }

    void handle_request()
    {
        http::response<http::string_body> res{
            http::status::ok,
            req_.version()
        };

        res.set(http::field::server, "Boost.Beast Multithreaded Server");
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.keep_alive(req_.keep_alive());

        if (req_.target() == "/")
        {
            res.body() = "Hello from the multithreaded Beast server\n";
        }
        else if (req_.target() == "/health")
        {
            res.body() = "OK\n";
        }
        else
        {
            res.result(http::status::not_found);
            res.body() = "Not found\n";
        }

        res.prepare_payload();
```

```
93
94          res_ = std::make_shared<http::response<http::string_body>>(std::move(res));
95
96          http::async_write(
97              stream_,
98              *res_,
99              asio::bind_executor(
100                 strand_,
101                 beast::bind_front_handler(
102                     &http_session::on_write,
103                     shared_from_this(),
104                     res_->need_eof())));
105     }
106
107     void on_write(bool close, beast::error_code ec, std::size_t)
108     {
109         if (ec)
110         {
111             std::cerr << "write error: " << ec.message() << '\n';
112             return;
113         }
114
115         res_.reset();
116
117         if (close)
118         {
119             do_close();
120             return;
121         }
```

```
122
123         do_read();
124     }
125
126     void do_close()
127     {
128         beast::error_code ec;
129         stream_.socket().shutdown(tcp::socket::shutdown_send, ec);
130     }
131
132 private:
133     beast::tcp_stream stream_;
134     asio::strand<asio::any_io_executor> strand_;
135     beast::flat_buffer buffer_;
136     http::request<http::string_body> req_;
137     std::shared_ptr<http::response<http::string_body>> res_;
138 };
139
140 class listener : public std::enable_shared_from_this<listener>
141 {
142 public:
143     listener(asio::io_context& io, tcp::endpoint endpoint)
144         : acceptor_(asio::make_strand(io))
145     {
146         beast::error_code ec;
147
148         acceptor_.open(endpoint.protocol(), ec);
149         if (ec)
150             throw beast::system_error(ec);
```

```
151
152         acceptor_.set_option(asio::socket_base::reuse_address(true), ec);
153         if (ec)
154             throw beast::system_error(ec);
155
156         acceptor_.bind(endpoint, ec);
157         if (ec)
158             throw beast::system_error(ec);
159
160         acceptor_.listen(asio::socket_base::max_listen_connections, ec);
161         if (ec)
162             throw beast::system_error(ec);
163     }
164
165     void run()
166     {
167         do_accept();
168     }
169
170 private:
171     void do_accept()
172     {
173         acceptor_.async_accept(
174             asio::make_strand(acceptor_.get_executor()),
175             beast::bind_front_handler(
176                 &listener::on_accept,
177                 shared_from_this()));
178     }
179
```

```
    void on_accept(beast::error_code ec, tcp::socket socket)
    {
        if (!ec)
        {
            std::make_shared<http_session>(std::move(socket))->run();
        }
        else
        {
            std::cerr << "accept error: " << ec.message() << '\n';
        }

        do_accept();
    }

private:
    tcp::acceptor acceptor_;
};

int main()
{
    try
    {
        const auto address = asio::ip::make_address("0.0.0.0");
        const unsigned short port = 8080;

        const std::size_t thread_count =
            std::max<std::size_t>(1, std::thread::hardware_concurrency());

        asio::io_context io(static_cast<int>(thread_count));
```

```
209
210         std::make_shared<listener>(
211             io,
212             tcp::endpoint(address, port))->run();
213
214         std::vector<std::thread> workers;
215         workers.reserve(thread_count - 1);
216
217         for (std::size_t i = 0; i + 1 < thread_count; ++i)
218         {
219             workers.emplace_back([&io]()
220             {
221                 io.run();
222             });
223         }
224
225         std::cout <<
            ↪ "Multithreaded HTTP server listening on http://127.0.0.1:8080\n";
226         std::cout << "Worker threads: " << thread_count << '\n';
227
228         io.run();
229
230         for (auto& t : workers)
231             t.join();
232     }
233     catch (const std::exception& ex)
234     {
235         std::cerr << "fatal error: " << ex.what() << '\n';
236         return EXIT_FAILURE;
```

```
    }

    return EXIT_SUCCESS;
}
```

16.7.2 Why this version is performance-ready

This version is performance-ready in the practical sense that it already uses the standard high-level scaling architecture:

- one shared asynchronous engine,
- one accept loop,
- one session per connection,
- multiple worker threads,
- strand-based per-session serialization,
- keep-alive request loop.

This is not the absolute end of optimization, but it is the correct architectural baseline for a serious multithreaded Beast server on Windows.

16.7.3 What still matters for real production performance

Even with the right threading model, real production performance still depends on additional factors such as:

- request parsing cost,
- response generation cost,

- logging overhead,
- allocator behavior,
- body size and payload copying,
- kernel socket limits,
- TLS cost if HTTPS is added,
- upstream database or filesystem latency.

The thread-pool model solves the core CPU-scaling problem at the networking layer, but it does not remove all other bottlenecks.

16.8 Practical Windows notes

16.8.1 Why this design fits Windows well

This design is well suited to Windows because Asio abstracts the platform-specific I/O completion mechanisms behind the same `io_context` model. The application code therefore scales in the same structural way as on other supported systems, while still using the standard Windows toolchains.

16.8.2 Typical MSVC build

Assuming Boost headers are installed at:

```
C:\Libraries\boost_1_89_0
```

a typical MSVC Developer Command Prompt build command is:

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 multithreaded_server.cpp
```

16.8.3 Typical MinGW-w64 build

A typical MinGW-w64 build command is:

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 multithreaded_server.cpp
↪ -o multithreaded_server.exe
```

16.8.4 Simple local tests

Useful local tests include:

```
curl http://127.0.0.1:8080/
curl http://127.0.0.1:8080/health
curl http://127.0.0.1:8080/not-found
```

For load testing, many simultaneous requests should be sent from a benchmark tool or script so that multiple worker threads become active.

16.9 Final reusable pattern

16.9.1 The architecture in one sentence

A scalable Boost.Beast server uses one shared `io_context` executed by multiple threads, one listener to accept connections asynchronously, and one strand-serialized session per connection.

16.9.2 The reusable pattern

The most reusable multithreaded pattern is:

1. create one `io_context`,

2. start the Beast listener on that context,
3. run the context from multiple worker threads,
4. associate accepted sockets with strand-aware executors,
5. serialize per-session handlers through a strand,
6. let independent sessions scale naturally across the thread pool.

16.10 Key takeaways

- Multiple threads may safely call `io_context::run()` to form the server's worker pool.
- The standard scaling pattern is one shared `io_context`, not one `io_context` per client.
- Strands are the core tool for protecting per-session state from concurrent handler execution.
- A multithreaded Beast server scales across sessions, while each session keeps its own logic sequential.
- A listener plus strand-based session design is the practical baseline for a performance-ready Windows server.

Part VII

Production Notes (Short & Practical)

Project Structure (Windows)

17.1 Introduction

A backend server can begin as a single source file, but production work becomes much easier when the project is organized from the start around a predictable directory layout, a clean CMake structure, and a compiler setup that behaves consistently on Windows.
For Boost.Asio and Boost.Beast projects, this matters even more because the codebase usually grows across several technical layers:

- transport and session handling,
- HTTP or WebSocket protocol code,
- routing and middleware,
- domain logic,
- storage and utility code.

If these concerns are mixed into one translation unit, maintenance becomes difficult very quickly. A practical Windows project therefore needs three things:

1. a stable CMake layout,

2. a clear source and header organization,
3. build settings that work well with MSVC and Clang-based toolchains.

This chapter focuses only on those practical production notes.

17.2 CMake layout

17.2.1 Why CMake is the right default

CMake is the standard cross-platform generator-based build system for modern C++ projects. The project is described with `CMakeLists.txt` files, and CMake generates the native buildsystem for the chosen generator on the machine. On Windows, that usually means Visual Studio, Ninja, or Ninja Multi-Config.
For a Boost.Asio and Boost.Beast backend project, CMake is especially useful because it gives a clean place to define:

- C++ standard level,
- include directories,
- target-specific compile options,
- source grouping,
- output directories,
- compiler-specific adjustments.

17.2.2 Recommended top-level layout

A practical project tree is:

```
ModernBackend/
 CMakeLists.txt
 cmake/
   warnings.cmake
 include/
   backend/
      app.hpp
      router.hpp
      session.hpp
      listener.hpp
      http_handlers.hpp
      websocket_session.hpp
      middleware.hpp
      store.hpp
      util.hpp
 src/
   main.cpp
   app.cpp
   router.cpp
   session.cpp
   listener.cpp
   http_handlers.cpp
   websocket_session.cpp
   middleware.cpp
   store.cpp
   util.cpp
```

```
examples/
  echo_server.cpp
  rest_api.cpp
  websocket_chat.cpp
tests/
  test_router.cpp
  test_store.cpp
  test_json.cpp
build/
```

This layout is not mandatory, but it is a very strong baseline for a Windows backend codebase.

17.2.3 Why separate `include/` and `src/`

Separating public headers from source files has several benefits:

- interfaces become easier to scan,
- dependencies become easier to reason about,
- test targets can include only the headers they need,
- the codebase scales more naturally when libraries are added later.

For a project that may later expose internal reusable components such as a router, middleware layer, or session abstraction, this separation is especially valuable.

17.2.4 Recommended top-level `CMakeLists.txt`

A practical minimal top-level configuration for a Windows Boost.Asio/Beast server is:

```
cmake_minimum_required(VERSION 3.21)

project(ModernBackend
    VERSION 1.0.0
    DESCRIPTION "Modern C++ Backend Engineering with Boost.Asio on Windows"
    LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

option(BACKEND_BUILD_EXAMPLES "Build example programs" ON)
option(BACKEND_BUILD_TESTS "Build tests" OFF)

add_executable(modern_backend
    src/main.cpp
    src/app.cpp
    src/router.cpp
    src/session.cpp
    src/listener.cpp
    src/http_handlers.cpp
    src/middleware.cpp
    src/store.cpp
    src/util.cpp
)

target_include_directories(modern_backend
    PRIVATE
        ${CMAKE_CURRENT_SOURCE_DIR}/include
```

```
)

find_package(Boost REQUIRED)

target_link_libraries(modern_backend
    PRIVATE
        Boost::boost
)

target_compile_features(modern_backend PRIVATE cxx_std_20)

if(MSVC)
    target_compile_options(modern_backend PRIVATE /W4 /permissive- /EHsc)
else()
    target_compile_options(modern_backend PRIVATE -Wall -Wextra -Wpedantic)
endif()

if(BACKEND_BUILD_EXAMPLES)
    add_subdirectory(examples)
endif()

if(BACKEND_BUILD_TESTS)
    enable_testing()
    add_subdirectory(tests)
endif()
```

17.2.5 Why `Boost::boost` is enough in many cases

Boost.Asio is header-only by default, and Boost.Beast is a header-only library built on Boost.Asio. Therefore many server projects that use Beast and Asio in the common header-only form can build by linking only the header-only Boost target.
This is one of the reasons Boost.Beast projects are convenient to structure in CMake: the integration can remain very lightweight for many applications.

17.2.6 When to split into an internal library

Once the backend grows beyond a few files, it is usually better to move the reusable server code into an internal library and keep the executable target very small.
A practical pattern is:

```
add_library(backend_core
    src/app.cpp
    src/router.cpp
    src/session.cpp
    src/listener.cpp
    src/http_handlers.cpp
    src/middleware.cpp
    src/store.cpp
    src/util.cpp
)

target_include_directories(backend_core
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/include
)

```

```
find_package(Boost REQUIRED)

target_link_libraries(backend_core
    PUBLIC
        Boost::boost
)

target_compile_features(backend_core PUBLIC cxx_std_20)

if(MSVC)
    target_compile_options(backend_core PRIVATE /W4 /permissive- /EHsc)
else()
    target_compile_options(backend_core PRIVATE -Wall -Wextra -Wpedantic)
endif()

add_executable(modern_backend src/main.cpp)
target_link_libraries(modern_backend PRIVATE backend_core)
```

This structure is stronger because:

- the server logic becomes reusable,
- tests can link to `backend_core`,
- examples can reuse the same core code,
- `main.cpp` stays focused on startup only.

17.2.7 Examples subdirectory

A clean `examples/CMakeLists.txt` may look like this:

```
add_executable(example_echo_server echo_server.cpp)
target_link_libraries(example_echo_server PRIVATE backend_core)

add_executable(example_rest_api rest_api.cpp)
target_link_libraries(example_rest_api PRIVATE backend_core)

add_executable(example_websocket_chat websocket_chat.cpp)
target_link_libraries(example_websocket_chat PRIVATE backend_core)
```

This is useful when the book's chapter examples are preserved as runnable programs alongside the main project.

17.2.8 Tests subdirectory

A practical `tests/CMakeLists.txt` can begin simply:

```
add_executable(test_router test_router.cpp)
target_link_libraries(test_router PRIVATE backend_core)
add_test(NAME test_router COMMAND test_router)

add_executable(test_store test_store.cpp)
target_link_libraries(test_store PRIVATE backend_core)
add_test(NAME test_store COMMAND test_store)
```

Even without an external test framework, this layout already helps enforce structure.

17.2.9 Do not misuse `CMAKE_BUILD_TYPE` on multi-config generators

On Windows, this is one of the most common CMake mistakes. Visual Studio and Ninja Multi-Config are multi-configuration generators. For those generators, the active configuration is chosen at build time, and `CMAKE_BUILD_TYPE` is ignored.

That means a project should not rely on single-config assumptions when targeting Visual Studio. A practical consequence is that configuration-specific behavior should be expressed with target properties and generator expressions rather than hard-coded single-config logic.

17.2.10 Useful output-directory layout

For Windows projects, it is often convenient to keep build outputs organized by configuration:

```
set_target_properties(modern_backend PROPERTIES
    RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin
    RUNTIME_OUTPUT_DIRECTORY_DEBUG ${CMAKE_BINARY_DIR}/bin/Debug
    RUNTIME_OUTPUT_DIRECTORY_RELEASE ${CMAKE_BINARY_DIR}/bin/Release
)
```

This avoids confusion when switching between Debug and Release builds in Visual Studio or Ninja Multi-Config.

17.3 File organization

17.3.1 Recommended code-level separation

A backend server becomes easier to maintain when code is grouped by responsibility rather than by chapter history. A practical grouping is:

- **bootstrap**: startup and configuration,
- **network**: listener, session, accept loop, socket code,
- **http**: request dispatch, response helpers, route handling,
- **websocket**: real-time session code,

- **middleware**: cross-cutting request pipeline,
- **store**: in-memory or database-backed resource handling,
- **util**: parsing, string helpers, JSON helpers, common utilities.

17.3.2 A practical header split

A strong header split for a Beast-based HTTP project is:

```
include/backend/
 app.hpp
 config.hpp
 listener.hpp
 http_session.hpp
 websocket_session.hpp
 router.hpp
 middleware.hpp
 responses.hpp
 request_utils.hpp
 json_utils.hpp
 user_store.hpp
 logging.hpp
```

This keeps each unit focused and prevents one monolithic header from becoming the center of the whole project.

17.3.3 What should stay in `main.cpp`

A practical rule is that `main.cpp` should do very little:

1. parse startup settings,

2. construct the application object,
3. create the `io_context`,
4. create the listener,
5. launch worker threads if needed,
6. call `run()`.

If `main.cpp` also contains route logic, storage logic, middleware logic, and session logic, the project has usually outgrown its current organization.

17.3.4 What belongs in the listener files

The listener unit should contain:

- acceptor setup,
- endpoint binding,
- listen call,
- asynchronous accept loop,
- creation of per-connection sessions.

It should not contain endpoint-specific business logic.

17.3.5 What belongs in the session files

The HTTP session unit should contain:

- request read loop,

- buffer ownership,
- timeout integration,
- write sequencing,
- keep-alive behavior,
- connection close behavior.

It should usually call outward into router or handler code, rather than embedding route-specific decisions itself.

17.3.6 What belongs in router and handler files

The router and handlers should contain:

- method dispatch,
- path matching,
- endpoint registration,
- request-to-response logic,
- validation and response formatting.

This layer should operate on request and response objects, not on raw sockets.

17.3.7 Why utility code should stay narrow

A common project-structure mistake is to create a vague `common.cpp` or `helpers.cpp` that grows without discipline. A better practice is to keep utility files narrow and named by purpose, such as:

- `json_utils.cpp`,
- `request_utils.cpp`,
- `string_utils.cpp`.

This makes dependencies easier to understand.

17.3.8 Recommended include style

A practical include style is:

```
#include <backend/router.hpp>
#include <backend/http_session.hpp>
#include <backend/user_store.hpp>
```

This is clearer than relying on fragile relative include paths deep inside the tree.

17.3.9 Suggested production split for a REST server

For a REST API project, a good file split is:

```
include/backend/
 app.hpp
 listener.hpp
 http_session.hpp
 router.hpp
 responses.hpp
 json_utils.hpp
 user.hpp
 user_store.hpp
```

```
src/
 main.cpp
 app.cpp
 listener.cpp
 http_session.cpp
 router.cpp
 responses.cpp
 json_utils.cpp
 user_store.cpp
```

This is already enough for a maintainable Windows backend service.

17.4 Build tips (MSVC/Clang)

17.4.1 General compiler strategy on Windows

On Windows, the most practical toolchain choices for this kind of project are usually:

- MSVC through the Visual Studio generator,
- MSVC through Ninja,
- Clang targeting the MSVC ABI through CMake generators such as Ninja or Visual Studio.

The key production principle is that build settings should be expressed per target, not as uncontrolled global flags.

17.4.2 Prefer target-based settings

A practical CMake rule is:

Set include directories, compile features, warnings, and runtime options on targets rather than through global directory-wide flags whenever possible.

This keeps the build more predictable, especially once examples and tests are added.

17.4.3 Recommended MSVC compile settings

For MSVC, a strong baseline is:

```
if(MSVC)
    target_compile_options(backend_core PRIVATE
        /W4
        /permissive-
        /EHsc
        /Zc:__cplusplus
    )
endif()
```

These options are practical because:

- `/W4` enables a strong warning baseline,
- `/permissive-` requests more standard-conforming behavior,
- `/EHsc` is the normal C++ exception model for this style of code,
- `/Zc:__cplusplus` improves the reported `__cplusplus` value for standards-aware code.

17.4.4 MSVC runtime selection

On Windows, runtime selection should use the CMake `MSVC_RUNTIME_LIBRARY` target property or its initializing variable rather than old ad hoc compiler-flag editing.
A practical modern setting is:

```
set_property(TARGET backend_core PROPERTY
    MSVC_RUNTIME_LIBRARY "MultiThreaded$<$<CONFIG:Debug>:Debug>DLL")
```

This is a clean way to express the common dynamic-runtime choice:

- Debug `/MDd`
- Release `/MD`

If a static runtime is desired, the value can be changed accordingly.

17.4.5 Why generator expressions are useful here

Generator expressions are evaluated per configuration and per toolchain context during generation. They are especially useful on Windows because Debug and Release are commonly built from the same multi-config tree.
That makes them a strong tool for configuration-specific options without breaking multi-config correctness.

17.4.6 Recommended Clang-on-Windows settings

When using Clang on Windows through CMake, a practical baseline is:

```
if(CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    target_compile_options(backend_core PRIVATE
        -Wall
        -Wextra
        -Wpedantic
    )
endif()
```

For Clang targeting the MSVC ABI, some MSVC-style flags may still matter depending on the selected generator and frontend mode, but the core project structure should stay the same.

17.4.7 Keep warnings high, but controlled

A practical production rule is:

- keep warnings high on your own code,
- do not turn every third-party warning into a project-wide problem,
- apply strict settings mainly to your own targets.

This matters with Boost-based projects because the application should remain strict without forcing awkward workarounds on every external header.

17.4.8 Use out-of-source builds

A Windows backend project should always use out-of-source builds. That means source files stay clean, while generated build files go into a separate directory such as:

```
build/
build-vs2022/
build-clang/
```

This is especially helpful when comparing MSVC and Clang builds side by side.

17.4.9 Typical configure commands

For Visual Studio 2022:

```
cmake -S . -B build-vs2022 -G "Visual Studio 17 2022"
cmake --build build-vs2022 --config Release
```

For Ninja with MSVC:

```
cmake -S . -B build-ninja-msvc -G Ninja
cmake --build build-ninja-msvc
```

For Ninja with Clang:

```
cmake -S . -B build-clang -G Ninja -D CMAKE_CXX_COMPILER=clang++
cmake --build build-clang
```

17.4.10 Do not assume one generator model

On Windows, Visual Studio and Ninja Multi-Config are multi-config, while plain Ninja is commonly single-config. A production-friendly project should avoid writing CMake logic that accidentally works only under one of those models.

17.4.11 Export compile commands when useful

When using editors, static analysis, or clang-based tooling, it is often useful to enable the compile command database:

```
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
```

This is especially useful in Ninja-based builds and increasingly useful in multi-config workflows as well.

17.4.12 Use target compile features instead of raw flags for the language level

Instead of hard-coding only compiler switches for the standard level, prefer expressing the requirement at the target level:

```
target_compile_features(backend_core PUBLIC cxx_std_20)
```

This keeps the project more portable and lets CMake choose the correct compiler flags for the active toolchain.

17.4.13 Practical note about Boost.Asio separate compilation

Boost.Asio is header-only by default, but it also supports a separate-compilation mode if a project deliberately chooses that model. For most Beast-based backend projects in this book's style, the default header-only approach is the simpler and more practical baseline.

17.4.14 Practical note about header selection

Boost documentation also notes that developers may include only the headers they need instead of always including umbrella headers. For very large projects, narrower includes can improve clarity and sometimes compile times, though using `<boost/asio.hpp>` and `<boost/beast.hpp>` is still perfectly reasonable while the codebase is small.

17.5 A compact production-ready CMake example

17.5.1 Complete practical baseline

The following `CMakeLists.txt` is a good short production-oriented baseline for a Windows Beast server:

```
cmake_minimum_required(VERSION 3.21)

project(ModernBackend
    VERSION 1.0.0
    LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

```

```
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

find_package(Boost REQUIRED)

add_library(backend_core
    src/app.cpp
    src/router.cpp
    src/listener.cpp
    src/session.cpp
    src/http_handlers.cpp
    src/middleware.cpp
    src/store.cpp
    src/util.cpp
)

target_include_directories(backend_core
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/include
)

target_link_libraries(backend_core
    PUBLIC
        Boost::boost
)

target_compile_features(backend_core PUBLIC cxx_std_20)

if(MSVC)
    target_compile_options(backend_core PRIVATE
```

```
        /W4
        /permissive-
        /EHsc
        /Zc:__cplusplus
    )

    set_property(TARGET backend_core PROPERTY
        MSVC_RUNTIME_LIBRARY "MultiThreaded$<$<CONFIG:Debug>:Debug>DLL")
else()
    target_compile_options(backend_core PRIVATE
        -Wall
        -Wextra
        -Wpedantic
    )
endif()

add_executable(modern_backend src/main.cpp)
target_link_libraries(modern_backend PRIVATE backend_core)

set_target_properties(modern_backend PROPERTIES
    RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin
    RUNTIME_OUTPUT_DIRECTORY_DEBUG ${CMAKE_BINARY_DIR}/bin/Debug
    RUNTIME_OUTPUT_DIRECTORY_RELEASE ${CMAKE_BINARY_DIR}/bin/Release
)
```

17.5.2 Why this is a good baseline

This baseline is strong because it already includes:

- C++20 setup,

- target-based organization,
- internal library split,
- Boost discovery,
- MSVC runtime control,
- warning settings,
- clean output directories,
- support for editor tooling.

It is short enough to understand quickly and strong enough to support real backend growth.

17.6 Final practical recommendations

17.6.1 Keep the structure stable early

A backend project benefits when the source tree stabilizes early. Reorganizing after dozens of files have accumulated is always harder than starting with a sensible layout.

17.6.2 Separate transport from application logic

On Windows server projects built with Beast and Asio, one of the most important structural habits is to keep:

- listeners and sessions in the network layer,
- routing and middleware in the application layer,
- storage in the data layer.

This division makes long-term maintenance much easier.

17.6.3 Prefer multiple targets over one giant executable target

As the project grows, a small set of focused targets is usually better than one giant build unit. A practical combination is:

- one core library,
- one main executable,
- optional example executables,
- optional test executables.

17.6.4 Build with both MSVC and Clang when practical

On Windows, using both MSVC and Clang-based builds during development often catches different classes of warnings and portability issues. Even if the final deployment uses one compiler, validating the code under both can improve quality significantly.

17.7 Key takeaways

- A strong Windows backend project should separate source layout, build layout, and output layout clearly.
- CMake should be target-oriented from the beginning, especially for warnings, language features, and runtime settings.
- Visual Studio and Ninja Multi-Config are multi-config generators, so build logic should not rely on single-config assumptions.
- Boost.Asio and Boost.Beast are very convenient in CMake projects because the common usage model is header-only.

- A clean split between listener/session code, router/handler code, and storage code is one of the most valuable long-term project-structure decisions.

Common Mistakes (Real Bugs)

18.1 Introduction

In real-world Boost.Asio and Boost.Beast backend systems, most failures are not caused by complex logic, but by a small set of recurring mistakes related to asynchronous execution, memory lifetime, and concurrency.
These bugs often pass initial testing and only appear under load, making them dangerous in production environments.
This chapter focuses on four critical categories of real bugs:

- dangling handlers,
- double write,
- partial read bugs,
- thread-related issues.

18.2 Dangling handlers

18.2.1 The problem

Asynchronous operations return immediately, while their handlers execute later. Any referenced data must remain valid until the handler completes.
A common mistake:

```
void send_bad(tcp::socket& socket)
{
    std::string msg = "hello\n";

    boost::asio::async_write(
        socket,
        boost::asio::buffer(msg),
        [](boost::system::error_code, std::size_t) {});
}
```

18.2.2 Why it fails

The string `msg` is destroyed when the function returns, but the write operation is still pending.

18.2.3 Correct solution

```
void send_good(tcp::socket& socket)
{
    auto msg = std::make_shared<std::string>("hello\n");

    boost::asio::async_write(
        socket,
```

```
        boost::asio::buffer(*msg),
        [msg](boost::system::error_code, std::size_t) {});
}
```

18.2.4 Session lifetime issue

Incorrect:

```
socket_.async_read_some(
    boost::asio::buffer(data_),
    [this](boost::system::error_code ec, std::size_t n)
    {
        if (!ec)
            do_write(n);
    });
```

Correct:

```
auto self = shared_from_this();

socket_.async_read_some(
    boost::asio::buffer(data_),
    [self](boost::system::error_code ec, std::size_t n)
    {
        if (!ec)
            self->do_write(n);
    });
```

18.2.5 Rule

Any object used by an async operation must stay alive until its handler completes.

18.3 Double write

18.3.1 The problem

Starting multiple writes on the same socket without waiting for completion leads to undefined behavior.
Incorrect:

```
void deliver(std::string msg)
{
    boost::asio::async_write(
        socket_,
        boost::asio::buffer(msg),
        [](auto, auto){});
}
```

Multiple calls to `deliver()` may overlap.

18.3.2 Symptoms

- corrupted responses,
- interleaved bytes,
- protocol failures.

18.3.3 Correct solution: write queue

```
class session : public std::enable_shared_from_this<session>
{
public:
```

```
    void deliver(std::string msg)
    {
        bool in_progress = !queue_.empty();
        queue_.push_back(std::move(msg));

        if (!in_progress)
            do_write();
    }

private:
    void do_write()
    {
        auto self = shared_from_this();

        boost::asio::async_write(
            socket_,
            boost::asio::buffer(queue_.front()),
            [self](boost::system::error_code ec, std::size_t)
            {
                if (ec) return;

                self->queue_.pop_front();

                if (!self->queue_.empty())
                    self->do_write();
            });
    }

private:
```

```
    tcp::socket socket_;
    std::deque<std::string> queue_;
};
```

18.3.4 Rule

Only one write operation should be active per connection.

18.4 Partial read bugs

18.4.1 The problem

Assuming one read equals one complete message.

Incorrect:

```
socket_.async_read_some(
    boost::asio::buffer(data_),
    [self](auto ec, std::size_t n)
    {
        std::string msg(self->data_.data(), n);
        self->handle(msg);
    });
```

18.4.2 Why it fails

TCP is a byte stream:

- messages may be split,
- messages may be combined.

18.4.3 Correct solution: delimiter-based

```
boost::asio::async_read_until(
    socket_,
    buffer_,
    '\n',
    [self](auto ec, std::size_t)
    {
        std::istream is(&self->buffer_);
        std::string line;
        std::getline(is, line);
        self->process(line);
    });
```

18.4.4 Correct solution: fixed-size read

```
boost::asio::async_read(
    socket_,
    boost::asio::buffer(header_),
    [self](auto ec, std::size_t)
    {
        self->parse_header();
    });
```

18.4.5 Rule

Never assume a read returns a complete logical message.

18.5 Thread issues

18.5.1 The problem

When multiple threads run `io_context::run()`, handlers may execute concurrently.

18.5.2 Incorrect assumption

Handlers will execute sequentially per session.
This is false in multithreaded environments.

18.5.3 Race condition example

```
void deliver(std::string msg)
{
    bool in_progress = !queue_.empty();
    queue_.push_back(std::move(msg));

    if (!in_progress)
        do_write();
}
```

If called from multiple threads, this causes data races.

18.5.4 Correct solution: strand

```
class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
```

```
        : socket_(std::move(socket)),
          strand_(socket_.get_executor())
    {
    }

    void deliver(std::string msg)
    {
        auto self = shared_from_this();

        boost::asio::post(
            strand_,
            [self, msg = std::move(msg)]() mutable
            {
                bool in_progress = !self->queue_.empty();
                self->queue_.push_back(std::move(msg));

                if (!in_progress)
                    self->do_write();
            });
    }

private:
    void do_write()
    {
        auto self = shared_from_this();

        boost::asio::async_write(
            socket_,
            boost::asio::buffer(queue_.front()),
```

```
            boost::asio::bind_executor(
                strand_,
                [self](auto ec, std::size_t)
                {
                    if (ec) return;

                    self->queue_.pop_front();

                    if (!self->queue_.empty())
                        self->do_write();
                }));
    }

private:
    tcp::socket socket_;
    boost::asio::strand<boost::asio::any_io_executor> strand_;
    std::deque<std::string> queue_;
};
```

18.5.5 Rule

Use a strand to serialize access to session state in multithreaded servers.

18.6 Key takeaways

- Keep all async-related data alive until handler completion.
- Never perform overlapping writes on the same connection.
- Always handle partial reads using a proper framing strategy.

- Use strands to prevent race conditions in multithreaded execution.

Conclusion

What you can build next

This volume began with the smallest meaningful asynchronous examples and gradually expanded toward practical backend systems using `Boost.Asio` and `Boost.Beast`. By the end of the journey, the reader has already worked through the core patterns required for real network software on Windows:

- event-driven execution with `io_context`,
- TCP session design,
- correct read and write patterns,
- timeout handling,
- HTTP request and response processing,
- JSON API construction,
- coroutine-based request flow,
- lightweight routing and middleware,
- REST-style services,

- WebSocket communication,
- multithreaded scaling with worker threads and strands.

That foundation is not academic. It is already enough to build useful backend systems that solve real operational problems.

A reader who understands the material in this book can move directly toward a wide range of production-oriented projects. Among the most practical next systems are internal service APIs. These are often the most valuable first deployment targets because they combine manageable complexity with immediate business usefulness. An internal service API may expose health endpoints, CRUD operations, job control, reporting status, or administrative automation for desktop or server-side applications. Such systems benefit greatly from the strengths of modern C++ backend code: explicit control, low overhead, predictable execution, and direct access to system-level integration points.

A second natural direction is a full REST backend with authentication and persistent storage. The in-memory examples shown earlier can be evolved into a practical service by replacing temporary storage with a real database layer, adding structured validation, improving request logging, and defining a cleaner boundary between transport code and domain logic. This path is especially appropriate for developers who want to build internal tooling, private APIs, industrial systems, or application backends where performance and deployment control matter more than framework fashion.

A third direction is real-time communication. The WebSocket material in this book already demonstrates the essential architecture for persistent connections, broadcast flows, and chat-like systems. From there, it becomes realistic to build dashboards, operator consoles, monitoring frontends, collaborative interfaces, message relays, multiplayer control channels, and low-latency device interfaces. The official Beast examples themselves demonstrate that HTTP and WebSocket patterns can coexist cleanly in one server, including multithreaded chat-oriented designs. That makes the transition from learning example to deployable service much shorter than many programmers initially assume.

Another important next step is coroutine-first backend design. Once the callback patterns are understood, C++20 coroutine support in Asio allows the same server ideas to be rewritten in a more linear and maintainable style. This does not remove the need for discipline, but it does make complex request flow easier to express. For larger APIs, protocol services, and structured request pipelines, this is often the point where the codebase starts to feel like a modern backend platform rather than a low-level networking experiment.
The reader can also move beyond plain HTTP services toward a more complete application platform. A small custom framework built on Beast can grow incrementally through additions such as:

- grouped route prefixes,
- request and response wrapper types,
- middleware composition,
- authentication and authorization layers,
- configuration loading,
- structured error responses,
- JSON helpers,
- WebSocket hubs,
- background task dispatch,
- persistent logging and metrics.

This path is particularly attractive because it allows the programmer to keep architectural control. Instead of inheriting the conventions and constraints of a heavy external framework, the programmer can grow only the abstractions that are actually needed.

A further practical direction is service specialization. Modern C++ backend software is particularly strong when the service is close to infrastructure, devices, protocols, binary data, or performance-sensitive workloads. Examples include:

- gateways between legacy TCP protocols and HTTP APIs,
- telemetry collectors,
- data ingestion services,
- command relays,
- low-latency control systems,
- binary protocol adapters,
- embedded management endpoints,
- administrative APIs for native desktop products,
- local service daemons on Windows.

These project classes often fit C++ unusually well because they combine network control, systems integration, memory awareness, and protocol precision.
The strongest lesson at this point is that the reader does not need to wait for a large team or a large framework before building serious backend software. The official Asio and Beast ecosystem already provides the transport and protocol building blocks, while modern C++ provides the language tools required for correctness, structure, and scale. What comes next is mainly a matter of choosing a domain, defining an architecture, and applying the patterns carefully.

Why C++ backend is underrated

C++ backend development is often underestimated not because it lacks capability, but because its strengths are different from the strengths usually emphasized in mainstream web discussions. Many conversations about backend programming focus first on rapid framework scaffolding, package ecosystems, or popularity of language-specific web stacks. Those are important concerns, but they do not define the full backend landscape.

When examined through the official Boost.Asio and Boost.Beast documentation and examples, modern C++ clearly offers a serious backend platform. Beast is explicitly positioned as a low-level, header-only foundation for HTTP/1 and WebSocket networking built on top of Asios asynchronous execution model. Asio itself provides the core execution model, timers, strands, thread-pool style scaling through multiple `io_context::run()` threads, and first-class C++20 coroutine support through `awaitable`, `use_awaitable`, and `co_spawn`. Those are not marginal capabilities. They are the core ingredients of modern backend engineering.

One reason C++ backend is underrated is that many developers still associate C++ primarily with desktop applications, game engines, embedded systems, compilers, or numerically intensive native code. Those associations are correct, but incomplete. The same properties that make C++ strong in those areas also make it a compelling backend language in certain classes of services: fine-grained control over memory and object lifetime, explicit concurrency architecture, predictable performance, strong type systems, RAII-based resource management, and direct integration with operating-system and protocol-level concepts.

Another reason is that C++ backend development demands more architectural awareness from the programmer. Framework-heavy ecosystems often provide immediate abstractions for routing, middleware, serialization, dependency injection, and deployment conventions. In C++, especially with Beast, the programmer sees the actual protocol and transport structure more directly. This can feel more demanding at first, but it also means the developer learns the real mechanics of the system rather than only learning framework usage. In the long term, that

usually leads to stronger engineering judgment.

C++ backend work is also underrated because many people assume that lower-level control necessarily means slow development. In reality, once a small internal framework layer is built, the development model becomes far more productive than outsiders often expect. The book itself demonstrates this progression: from raw `io_context` and timers to sessions, then to HTTP, routing, JSON, middleware, REST, WebSocket, coroutines, and multithreaded scaling. The progression shows that C++ backend code can become highly structured and expressive without giving up control over the transport and execution model.

There is also a strong practical argument in favor of C++ backend work on Windows. Many organizations already depend on native C++ applications, services, tools, devices, SDKs, or internal libraries. In such environments, a backend written in modern C++ can integrate more directly with the rest of the native stack than a separate runtime or framework layer might. This is especially true when the service must interact closely with local resources, binary formats, performance-sensitive paths, device communication, or native Windows APIs.

Another source of underestimation is confusion between *web development* and *backend engineering*. They overlap, but they are not identical. A large portion of backend engineering is not primarily about rendering HTML pages or following mainstream web trends. It is about protocol handling, message flow, concurrency, resource management, resilience, observability, and service boundaries. From that perspective, modern C++ with Asio and Beast is not an unusual choice at all. It is often a very rational choice.

The official Beast examples reinforce this point. The examples include HTTP servers, WebSocket servers, chat-oriented systems, asynchronous and multithreaded patterns, and practical combinations of technologies. This demonstrates that the library is intended not merely as a protocol parser, but as a real foundation for service construction. Official Asio support for coroutine-based asynchronous programming strengthens that further by making the code easier to maintain without abandoning the non-blocking execution model.

The real issue, therefore, is not whether C++ can build modern backends. It clearly can. The

more accurate question is whether the project benefits from the kinds of advantages C++ offers. If the service needs strong control, deep integration, predictable behavior, low overhead, custom protocol handling, or system-level proximity, then modern C++ backend engineering is not merely viable; it may be one of the best choices available.
For that reason, C++ backend development should not be viewed as a niche curiosity. It should be viewed as a serious engineering path for teams and individuals who value precision, efficiency, and architectural control.

Suggested next steps

A practical conclusion should not end only with encouragement. It should also provide a concrete path forward. The strongest next step after finishing this volume is to choose one complete project and build it end-to-end. The project does not need to be large. In fact, the best first production step is usually small enough to finish, test, and revise quickly.
A very strong first project is a complete internal REST API with:

- a listener,
- session objects,
- route registration,
- JSON parsing,
- in-memory or persistent storage,
- authentication,
- logging middleware,
- clean error responses,

- a small test client.

This project consolidates nearly every major idea from the book. It is also highly reusable because the resulting code can become the seed of future services.
A second recommended project is a WebSocket-based real-time system. The official Beast examples make clear that chat-like and broadcast-oriented systems are practical and well supported. A reader can extend the chapter material into a live monitoring server, event broadcaster, dashboard backend, or team messaging prototype. Such a project teaches not only transport handling but also session registries, shared state, write queues, and message fan-out patterns.
A third next step is architectural refactoring. Readers who have followed the examples chapter by chapter may now benefit from reorganizing the code into a production-style layout:

- `listener`,
- `http_session`,
- `websocket_session`,
- `router`,
- `middleware`,
- `responses`,
- `store`,
- `app`,
- `main`.

This is the point where the material becomes a real codebase rather than a sequence of examples.

Another excellent next step is to convert one callback-based service fully to coroutines. The official Asio coroutine facilities are strong enough to support serious asynchronous logic, and rewriting a working service in coroutine style is one of the best ways to understand both models deeply. A good exercise is to take a REST endpoint service or WebSocket session from the earlier chapters and re-express it entirely with `co_await`, while preserving the same semantics and correctness.

Readers interested in production hardening should then add the concerns that distinguish demos from durable services:

- connection timeouts,
- request size limits,
- structured logging,
- graceful shutdown,
- metrics,
- configuration files,
- defensive validation,
- thread-pool tuning,
- persistent storage,
- deployment packaging.

This is where backend engineering becomes operational engineering.

It is also highly recommended to study the official Beast examples directly after finishing this volume. They are valuable not only because they work, but because they demonstrate how the library authors structure real examples for HTTP, WebSocket, multithreading, and chat-style systems. Revisiting those examples after understanding the material in this book is far more productive than reading them too early. Many architectural choices that might seem opaque at first become much clearer after completing a structured journey through sessions, queues, strands, timeouts, and protocol flow.

A final recommended next step is personal specialization. After the broad base established by this volume, the reader should begin choosing a direction rather than staying permanently at the level of general examples. Practical specialization paths include:

- HTTP API systems,
- WebSocket real-time systems,
- custom TCP protocols,
- Windows-native service daemons,
- industrial and device gateways,
- internal enterprise tooling,
- high-performance microservices,
- protocol adapters and relays.

The purpose of this book was never merely to teach isolated Asio syntax or Beast calls. Its purpose was to establish a durable engineering mindset for modern C++ backend software on Windows. That mindset combines protocol awareness, correctness under asynchronous execution, disciplined state management, and controlled growth of abstractions.

If the reader now proceeds to build one real system, refactor it into a clean structure, test it under stress, and iterate until it feels stable, then this volume has already succeeded in its deeper goal.

The next chapter does not need to be another tutorial. The next chapter can be your own server.

Appendices

Appendix A — Full Reusable Code Snippets

This appendix collects compact, reusable code snippets based on the official Boost.Asio and Boost.Beast programming model: asynchronous execution with `io_context`, socket and timer operations, Beast HTTP and WebSocket message handling, strand-based serialization, and coroutine support through `awaitable` and `co_spawn`. The official Beast examples and Asio documentation consistently use these patterns as the foundation for practical servers, including HTTP sessions, WebSocket sessions, multithreaded execution, and coroutine-based request flow.

A.1 Minimal `io_context` and timer

The timer example is one of the smallest complete Asio programs because it demonstrates the full execution model in one file: construct an `io_context`, associate an asynchronous object with it, initiate an operation, and drive completion through `run()`. Official Asio tutorial material uses this structure to introduce the event-driven model.

```
#include <iostream>
#include <chrono>
#include <boost/asio.hpp>
```

```
4
5  namespace asio = boost::asio;
6  using boost::system::error_code;
7
8  int main()
9  {
10     try
11     {
12         asio::io_context io;
13
14         asio::steady_timer timer(io);
15         timer.expires_after(std::chrono::seconds(1));
16
17         timer.async_wait(
18             [](const error_code& ec)
19             {
20                 if (!ec)
21                     std::cout << "timer expired\n";
22             });
23
24         io.run();
25     }
26     catch (const std::exception& ex)
27     {
28         std::cerr << "fatal error: " << ex.what() << '\n';
29         return 1;
30     }
31
32     return 0;
```

```
}
```

A.2 Reusable TCP session skeleton

A session object is the standard reusable unit of per-client state in Asio-style servers. The session owns the socket, owns the input buffer, begins with `start()`, and preserves object lifetime with `std::enable_shared_from_this`. This is the core pattern repeated throughout official asynchronous server examples.

```
#include <array>
#include <iostream>
#include <memory>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class session : public std::enable_shared_from_this<session>
{
public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        do_read();
    }
```

```

private:
    void do_read()
    {
        auto self = shared_from_this();

        socket_.async_read_some(
            asio::buffer(data_),
            [self](error_code ec, std::size_t length)
            {
                if (ec)
                {
                    if (ec != asio::error::eof)
                        std::cerr << "read error: " << ec.message() << '\n';
                    return;
                }

                self->do_write(length);
            });
    }

    void do_write(std::size_t length)
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(data_.data(), length),
            [self](error_code ec, std::size_t)
```

```
            {
                if (ec)
                {
                    std::cerr << "write error: " << ec.message() << '\n';
                    return;
                }

                self->do_read();
            });
    }

private:
    tcp::socket socket_;
    std::array<char, 4096> data_{};
};
```

A.3 Reusable async accept loop

The accept loop is the core listener pattern. The server starts one asynchronous accept, creates a session when a client arrives, and immediately re-arms accept. This is the central scalable structure in official Asio and Beast server examples.

```
#include <iostream>
#include <memory>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

```

```
class server
{
public:
    server(asio::io_context& io, unsigned short port)
        : acceptor_(io, tcp::endpoint(tcp::v4(), port))
    {
    }

    void start()
    {
        do_accept();
    }

private:
    void do_accept()
    {
        acceptor_.async_accept(
            [this](error_code ec, tcp::socket socket)
            {
                if (!ec)
                    std::make_shared<session>(std::move(socket))->start();
                else
                    std::cerr << "accept error: " << ec.message() << '\n';

                do_accept();
            });
    }

private:
```

```
    tcp::acceptor acceptor_;
};
```

A.4 Safe write queue pattern

Official Asio/Beast practice strongly implies one of the most important real-world rules: do not initiate overlapping writes on the same stream without deliberate serialization. A per-session queue is the practical fix. This pattern keeps message storage alive and ensures that only one write operation is active at a time.

```
#include <deque>
#include <memory>
#include <string>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using boost::system::error_code;

class queued_writer : public std::enable_shared_from_this<queued_writer>
{
public:
    explicit queued_writer(asio::ip::tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void deliver(std::string msg)
    {
        bool write_in_progress = !queue_.empty();
        queue_.push_back(std::move(msg));
```

```
21
22          if (!write_in_progress)
23              do_write();
24      }
25
26  private:
27      void do_write()
28      {
29          auto self = shared_from_this();
30
31          asio::async_write(
32              socket_,
33              asio::buffer(queue_.front()),
34              [self](error_code ec, std::size_t)
35              {
36                  if (ec)
37                      return;
38
39                  self->queue_.pop_front();
40
41                  if (!self->queue_.empty())
42                      self->do_write();
43              });
44      }
45
46  private:
47      asio::ip::tcp::socket socket_;
48      std::deque<std::string> queue_;
49  };
```

A.5 Line-based protocol reader

Official Asio provides `async_read_until()` specifically for delimiter-based protocols. This is the correct reusable pattern when the server must accumulate bytes until a complete line becomes available. The dynamic buffer persists across reads and preserves unread data for future parsing.

```
#include <iostream>
#include <istream>
#include <memory>
#include <string>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class line_session : public std::enable_shared_from_this<line_session>
{
public:
    explicit line_session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        read_line();
    }

```

```
private:
    void read_line()
    {
        auto self = shared_from_this();

        asio::async_read_until(
            socket_,
            input_,
            '\n',
            [self](error_code ec, std::size_t)
            {
                if (ec)
                    return;

                std::istream is(&self->input_);
                std::string line;
                std::getline(is, line);

                self->handle_line(line);
                self->read_line();
            });
    }

    void handle_line(const std::string& line)
    {
        std::cout << "line: " << line << '\n';
    }

private:
```

```
    tcp::socket socket_;
    asio::streambuf input_;
};
```

A.6 Length-prefixed binary reader

A reliable binary protocol usually reads a fixed-size header first and then reads exactly the payload length. This pattern avoids partial-read parsing bugs and matches Asios intended use of `async_read()` for exact-size fields.

```
#include <array>
#include <cstdint>
#include <memory>
#include <vector>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class binary_session : public std::enable_shared_from_this<binary_session>
{
public:
    explicit binary_session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
```

```
        read_header();
    }

private:
    static std::uint32_t load_be32(const std::array<unsigned char, 4>& b)
    {
        return (static_cast<std::uint32_t>(b[0]) << 24) |
               (static_cast<std::uint32_t>(b[1]) << 16) |
               (static_cast<std::uint32_t>(b[2]) << 8)  |
               (static_cast<std::uint32_t>(b[3]));
    }

    void read_header()
    {
        auto self = shared_from_this();

        asio::async_read(
            socket_,
            asio::buffer(header_),
            [self](error_code ec, std::size_t)
            {
                if (ec)
                    return;

                self->payload_length_ = load_be32(self->header_);
                self->payload_.assign(self->payload_length_, 0);
                self->read_payload();
            });
    }
```

```

    void read_payload()
    {
        auto self = shared_from_this();

        asio::async_read(
            socket_,
            asio::buffer(payload_),
            [self](error_code ec, std::size_t)
            {
                if (ec)
                    return;

                self->process_message();
            });
    }

    void process_message()
    {
    }

private:
    tcp::socket socket_;
    std::array<unsigned char, 4> header_{};
    std::uint32_t payload_length_ = 0;
    std::vector<unsigned char> payload_;
};
```

A.7 Session timeout with `steady_timer`

Official timer semantics in Asio are central to practical session stability: `expires_after()` resets the deadline, pending waits can complete with `operation_aborted`, and stalled socket I/O is commonly terminated by closing the socket. This snippet shows the standard reusable idle-timeout pattern.

```
#include <array>
#include <chrono>
#include <memory>
#include <boost/asio.hpp>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class timed_session : public std::enable_shared_from_this<timed_session>
{
public:
    explicit timed_session(tcp::socket socket)
        : socket_(std::move(socket)),
          timer_(socket_.get_executor())
    {
    }

    void start()
    {
        arm_timeout();
        do_read();
    }
```

```
private:
    void arm_timeout()
    {
        auto self = shared_from_this();

        timer_.expires_after(std::chrono::seconds(30));

        timer_.async_wait(
            [self](error_code ec)
            {
                if (ec == asio::error::operation_aborted)
                    return;

                if (!ec)
                    self->close();
            });
    }

    void do_read()
    {
        auto self = shared_from_this();

        socket_.async_read_some(
            asio::buffer(data_),
            [self](error_code ec, std::size_t)
            {
                if (ec)
                {
```

```
                    self->close();
                    return;
                }

                self->arm_timeout();
                self->do_read();
            });
    }

    void close()
    {
        error_code ignored_ec;
        timer_.cancel(ignored_ec);
        socket_.shutdown(tcp::socket::shutdown_both, ignored_ec);
        socket_.close(ignored_ec);
    }

private:
    tcp::socket socket_;
    asio::steady_timer timer_;
    std::array<char, 4096> data_{};
};
```

A.8 Minimal Beast HTTP session

Official Beast examples consistently use a persistent `flat_buffer`, a parsed HTTP request object, a typed HTTP response, and `http::read`/`http::write` or their asynchronous variants. This is the smallest reusable synchronous session pattern.

```
#include <boost/asio.hpp>
```

```
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <iostream>

namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

void handle_http_session(tcp::socket socket)
{
    try
    {
        beast::flat_buffer buffer;
        http::request<http::string_body> req;

        http::read(socket, buffer, req);

        http::response<http::string_body> res{
            http::status::ok,
            req.version()
        };

        res.set(http::field::server, "Beast HTTP Server");
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.keep_alive(false);
        res.body() = "Hello HTTP\n";
        res.prepare_payload();

```

```
        http::write(socket, res);

        beast::error_code ec;
        socket.shutdown(tcp::socket::shutdown_send, ec);
    }
    catch (const std::exception& ex)
    {
        std::cerr << "session error: " << ex.what() << '\n';
    }
}
```

A.9 Minimal HTTP router

Beast provides the HTTP message layer, while routing remains application code built on top of `req.method()` and `req.target()`. A compact exact-match router is a strong reusable baseline.

```
#include <boost/beast/http.hpp>
#include <functional>
#include <map>
#include <string>
#include <utility>

namespace http = boost::beast::http;

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;
using handler_type  = std::function<response_type(const request_type&)>;
using route_key     = std::pair<http::verb, std::string>;

response_type make_text_response(
```

```
    const request_type& req,
    http::status status,
    std::string body)
{
    response_type res{status, req.version()};
    res.set(http::field::server, "Router");
    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.keep_alive(req.keep_alive());
    res.body() = std::move(body);
    res.prepare_payload();
    return res;
}

class router
{
public:
    void get(std::string path, handler_type handler)
    {
        routes_[{http::verb::get, std::move(path)}] = std::move(handler);
    }

    void post(std::string path, handler_type handler)
    {
        routes_[{http::verb::post, std::move(path)}] = std::move(handler);
    }

    response_type route(const request_type& req) const
    {
        auto it = routes_.find({req.method(), std::string(req.target())});
```

```
        if (it != routes_.end())
            return it->second(req);

        return make_text_response(req, http::status::not_found, "Not found\n");
    }

private:
    std::map<route_key, handler_type> routes_;
};
```

A.10 Minimal JSON API helpers

Boost.JSON provides direct parse and serialize support for HTTP JSON bodies. A reusable API helper should centralize JSON responses and error responses to keep handlers compact.

```
#include <boost/beast/http.hpp>
#include <boost/json.hpp>
#include <string>

namespace http = boost::beast::http;
namespace json = boost::json;

using request_type  = http::request<http::string_body>;
using response_type = http::response<http::string_body>;

response_type make_json_response(
    const request_type& req,
    http::status status,
    const json::value& body)
{
```

```
    response_type res{status, req.version()};
    res.set(http::field::server, "JSON API");
    res.set(http::field::content_type, "application/json");
    res.keep_alive(req.keep_alive());
    res.body() = json::serialize(body);
    res.prepare_payload();
    return res;
}

response_type make_json_error(
    const request_type& req,
    http::status status,
    std::string message)
{
    json::object obj;
    obj["error"] = std::move(message);
    return make_json_response(req, status, obj);
}
```

A.11 Coroutine HTTP session

Official Asio coroutine support is built around `awaitable`, `use_awaitable`, and `co_spawn`. Official Beast examples also include awaitable HTTP examples, and the core reusable pattern is a linear request loop using `co_await http::async_read` and `co_await http::async_write`.

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <string>

```

```
namespace asio  = boost::asio;
namespace beast = boost::beast;
namespace http  = beast::http;
using tcp = asio::ip::tcp;

asio::awaitable<void> http_session(tcp::socket socket)
{
    beast::flat_buffer buffer;

    for (;;)
    {
        http::request<http::string_body> req;

        co_await http::async_read(
            socket,
            buffer,
            req,
            asio::use_awaitable);

        http::response<http::string_body> res{
            http::status::ok,
            req.version()
        };

        res.set(http::field::server, "Coroutine HTTP");
        res.set(http::field::content_type, "text/plain; charset=utf-8");
        res.keep_alive(req.keep_alive());
        res.body() = "Hello from coroutine session\n";
        res.prepare_payload();
```

```

        bool keep_alive = res.keep_alive();

        co_await http::async_write(
            socket,
            res,
            asio::use_awaitable);

        if (!keep_alive)
            break;
    }
}
```

A.12 Coroutine listener with co_spawn

This is the standard reusable accept-loop pattern for coroutine servers: accept a socket with `co_await`, then launch a detached per-connection coroutine using `co_spawn`. The official Asio coroutine documentation and Beast awaitable examples follow exactly this model.

```
#include <boost/asio.hpp>
#include <boost/beast/http.hpp>

namespace asio = boost::asio;
using tcp = asio::ip::tcp;

asio::awaitable<void> listener(tcp::acceptor acceptor)
{
    for (;;)
    {
        tcp::socket socket =
```

```
            co_await acceptor.async_accept(asio::use_awaitable);

        asio::co_spawn(
            acceptor.get_executor(),
            http_session(std::move(socket)),
            asio::detached);
    }
}
```

A.13 Minimal WebSocket echo session

Official Beast documentation and examples present WebSocket as a first-class protocol layer on top of Asio. The smallest reusable session performs the handshake, reads a frame into a persistent dynamic buffer, and writes the same frame back.

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/websocket.hpp>
#include <iostream>

namespace asio      = boost::asio;
namespace beast     = boost::beast;
namespace websocket = beast::websocket;
using tcp = asio::ip::tcp;

void websocket_echo_session(tcp::socket socket)
{
    try
    {
        websocket::stream<tcp::socket> ws(std::move(socket));
```

```
        ws.accept();

        beast::flat_buffer buffer;

        for (;;)
        {
            ws.read(buffer);
            ws.text(ws.got_text());
            ws.write(buffer.data());
            buffer.consume(buffer.size());
        }
    }
    catch (const std::exception& ex)
    {
        std::cerr << "websocket error: " << ex.what() << '\n';
    }
}
```

A.14 WebSocket upgrade from HTTP request

Official Beast WebSocket handshaking documentation explicitly shows the reusable pattern for servers that first parse HTTP themselves, then decide whether the request is a WebSocket upgrade. This is the right foundation for mixed HTTP/WebSocket servers.

```
#include <boost/asio.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <boost/beast/websocket.hpp>

namespace asio      = boost::asio;
```

```
namespace beast     = boost::beast;
namespace http      = beast::http;
namespace websocket = beast::websocket;
using tcp = asio::ip::tcp;

void handle_http_or_ws(tcp::socket socket)
{
    beast::flat_buffer buffer;
    http::request<http::string_body> req;

    http::read(socket, buffer, req);

    if (websocket::is_upgrade(req))
    {
        websocket::stream<tcp::socket> ws(std::move(socket));
        ws.accept(req);
        return;
    }

    http::response<http::string_body> res{
        http::status::ok,
        req.version()
    };

    res.set(http::field::content_type, "text/plain; charset=utf-8");
    res.body() = "ordinary HTTP response\n";
    res.prepare_payload();

    http::write(socket, res);
```

```
}
```

A.15 Multithreaded `io_context` worker pool

Official Asio documentation explicitly supports running one `io_context` from multiple threads. This is the core scaling pattern for multithreaded servers.

```
#include <boost/asio.hpp>
#include <thread>
#include <vector>

namespace asio = boost::asio;

int main()
{
    asio::io_context io;
    auto guard = asio::make_work_guard(io);

    std::vector<std::thread> workers;
    const std::size_t thread_count =
        std::max<std::size_t>(1, std::thread::hardware_concurrency());

    for (std::size_t i = 0; i < thread_count; ++i)
    {
        workers.emplace_back([&io]()
        {
            io.run();
        });
    }

```

```
    guard.reset();

    for (auto& t : workers)
        t.join();
}
```

A.16 Strand-serialized session for multithreaded servers

Once multiple threads call `io_context::run()`, session state must be protected from concurrent handler execution. The official Asio model uses strands for this. A reusable session should post queue mutation and bind write completions through the same strand.

```
#include <boost/asio.hpp>
#include <deque>
#include <memory>
#include <string>

namespace asio = boost::asio;
using asio::ip::tcp;
using boost::system::error_code;

class threaded_session : public std::enable_shared_from_this<threaded_session>
{
public:
    explicit threaded_session(tcp::socket socket)
        : socket_(std::move(socket)),
          strand_(socket_.get_executor())
    {
    }

```

```
    void deliver(std::string msg)
    {
        auto self = shared_from_this();

        asio::post(
            strand_,
            [self, msg = std::move(msg)]() mutable
            {
                bool write_in_progress = !self->queue_.empty();
                self->queue_.push_back(std::move(msg));

                if (!write_in_progress)
                    self->do_write();
            });
    }

private:
    void do_write()
    {
        auto self = shared_from_this();

        asio::async_write(
            socket_,
            asio::buffer(queue_.front()),
            asio::bind_executor(
                strand_,
                [self](error_code ec, std::size_t)
                {
                    if (ec)
```

```
                        return;

                    self->queue_.pop_front();

                    if (!self->queue_.empty())
                        self->do_write();
                }));
    }

private:
    tcp::socket socket_;
    asio::strand<asio::any_io_executor> strand_;
    std::deque<std::string> queue_;
};
```

A.17 Minimal CMake baseline

A practical Windows backend project using Boost.Asio and Beast often needs only the header-only Boost target. This compact CMake file is a reusable starting point for examples in this volume.

```
cmake_minimum_required(VERSION 3.21)

project(ModernBackend LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

find_package(Boost REQUIRED)
```

```
10
11 add_executable(modern_backend
12     src/main.cpp
13 )
14
15 target_link_libraries(modern_backend
16     PRIVATE
17         Boost::boost
18 )
19
20 target_include_directories(modern_backend
21     PRIVATE
22         ${CMAKE_CURRENT_SOURCE_DIR}/include
23 )
24
25 if(MSVC)
26     target_compile_options(modern_backend PRIVATE /W4 /permissive- /EHsc)
27 else()
28     target_compile_options(modern_backend PRIVATE -Wall -Wextra -Wpedantic)
29 endif()
```

A.18 Practical Windows compile commands

The examples in this book are intentionally designed to be easy to compile on Windows with either MSVC or MinGW-w64 when Boost headers are available locally.

MSVC

```
cl /EHsc /std:c++20 /W4 /I C:\Libraries\boost_1_89_0 example.cpp
```

MinGW-w64

```
g++ -std=c++20 -Wall -Wextra -I C:\Libraries\boost_1_89_0 example.cpp -o example.exe
```

A.19 Final practical note

The most reusable code in Boost.Asio and Boost.Beast is not the most complicated code. It is the code that preserves three disciplines consistently:

- correct object and buffer lifetime,
- one clear asynchronous flow per session,
- explicit serialization wherever concurrency could corrupt state.

The official Asio and Beast examples repeatedly reinforce these same principles across timers, sockets, HTTP servers, coroutine sessions, WebSocket servers, and multithreaded chat-style systems. This appendix therefore should be treated not merely as a collection of snippets, but as a compact map of the patterns that remain valid as projects become larger.

Appendix B — Debugging Checklist

This appendix provides a practical debugging checklist for modern backend applications built with `Boost.Asio` and `Boost.Beast` on Windows. The focus is not merely on fixing syntax errors or obvious crashes, but on diagnosing the kinds of faults that commonly appear in asynchronous systems: handlers not running, sockets closing unexpectedly, stalled requests, partial reads, lifetime bugs, race conditions, executor confusion, invalid assumptions about callback order, silent exception paths, and platform-specific failures related to Winsock or Visual Studio configuration.

In asynchronous C++ systems, debugging must be systematic. A backend built on `io_context`, timers, sockets, strands, and coroutine-based operations may compile cleanly and still fail at runtime because one invariant was broken: an object died too early, a handler was posted to the wrong execution context, a socket was never kept alive, an exception escaped from a completion path, or an error code was ignored. The goal of this checklist is to provide a repeatable workflow that can be used during development, before release, and while investigating production issues reproduced in a controlled Windows environment.

Core principle of backend debugging

Before starting any technical investigation, establish the most important debugging rule for asynchronous backend systems:

> Never debug symptoms in isolation. Always trace the failing request or connection through its full lifecycle: construction, initiation, execution context, callback or coroutine continuation, error handling, shutdown, and destruction.

A single visible symptom such as “client hangs” or “request never completes” can be caused by several very different issues:

- the `io_context` never runs,
- the operation was never started,
- the socket was closed from another path,
- the handler was serialized by a strand and is waiting behind another task,
- the request parser is waiting for more bytes,
- a timeout cancelled the operation,

- an exception aborted the session object,
- or a lifetime bug invalidated captured state.

For that reason, every debugging session should begin by reconstructing the execution path, not by guessing at the apparent failure point.

Build configuration checklist for Windows

Always begin with a debug-friendly build. Many backend issues become unnecessarily difficult when optimization, inlining, and stripped symbols hide the true call path.
Use a Visual Studio configuration that preserves symbols and predictable execution during debugging.

```
Recommended Debug configuration on Windows:

Configuration Type: Application or DLL as needed
C/C++ > General > Debug Information Format: Program Database (/Zi)
C/C++ > Optimization > Optimization: Disabled (/Od)
C/C++ > Code Generation > Runtime Library: Multi-threaded Debug DLL (/MDd)
C/C++ > Language > Enable C++ Exceptions: Yes (/EHsc)
Linker > Debugging > Generate Debug Info: Yes (/DEBUG:FULL)
```

If you are using sanitizers for memory validation, use a dedicated debug or diagnostic configuration:

```
C/C++ > General > Enable Address Sanitizer: Yes (/fsanitize=address)
```

For CMake-based projects on Windows, a comparable diagnostic preset can look like this:

```
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

if (MSVC)
    add_compile_options(/W4 /EHsc /Zi /Od /MDd)
    add_link_options(/DEBUG:FULL)
endif()

add_executable(server main.cpp)
target_link_libraries(server PRIVATE ws2_32)
```

Checklist:

- Confirm that the executable being run under the debugger is the same one just built.
- Confirm that the correct architecture is selected: usually `x64`.
- Confirm that debug symbols are loaded.
- Confirm that stale binaries from another build directory are not being launched.
- Confirm that the working directory matches the location expected by your configuration files, certificates, logs, and static files.

Startup validation checklist

If the server fails at startup, do not proceed to runtime investigation until startup has been validated step by step.
At process startup, verify the following:

- configuration file was opened successfully,

- logging sink was initialized,
- listening address was parsed correctly,
- listening port is valid and available,
- Winsock-related prerequisites are satisfied by the runtime environment,
- TLS files, if used, exist and are readable,
- thread pool size and execution model are as intended,
- the acceptor has been opened, bound, and marked listening,
- the `io_context` or equivalent execution engine is actually run.

A minimal startup checkpoint macro set may help during early development:

```
#include <iostream>
#include <string>
#include <system_error>

#define DBG_POINT(msg) \
    do { std::cerr << "[DBG] " << msg << '\n'; } while(false)

#define DBG_EC(where, ec) \
    do { \
        if (ec) { \
            std::cerr << "[ERR] " << where << ": " \
                      << ec.message() << " (" << ec.value() << ")\n"; \
        } \
    } while(false)
```

Use it during startup:

```
DBG_POINT("server startup begin");

boost::system::error_code ec;

tcp::endpoint ep{boost::asio::ip::make_address("0.0.0.0", ec), 8080};
DBG_EC("make_address", ec);

tcp::acceptor acceptor{ioc};
acceptor.open(ep.protocol(), ec);
DBG_EC("acceptor.open", ec);

acceptor.set_option(boost::asio::socket_base::reuse_address(true), ec);
DBG_EC("acceptor.set_option(reuse_address)", ec);

acceptor.bind(ep, ec);
DBG_EC("acceptor.bind", ec);

acceptor.listen(boost::asio::socket_base::max_listen_connections, ec);
DBG_EC("acceptor.listen", ec);

DBG_POINT("server startup end");
```

If startup fails, stop there and fix the first failing invariant. Do not move forward until bind, listen, and event-loop execution are confirmed.

Event loop checklist

A large percentage of “nothing happens” failures in `Boost.Asio` applications are caused by incorrect event loop handling. When a handler never executes, confirm these points first:

- Was `io_context::run()` called?
- Is it still running, or did it return immediately?
- Did all work finish before asynchronous operations were posted?
- Was a work guard required but missing?
- Was the handler posted to a different executor than expected?
- Is another thread blocked while holding a required resource?

A simple debug pattern for checking event loop activity:

```
#include <boost/asio.hpp>
#include <iostream>

namespace asio = boost::asio;

int main() {
    asio::io_context ioc;

    asio::post(ioc, [] {
        std::cerr << "[DBG] first posted task executed\n";
    });

    std::cerr << "[DBG] before run()\n";
    auto count = ioc.run();
    std::cerr << "[DBG] after run(), executed handlers = " << count << '\n';
}
```

If `run()` returns immediately in a server application, investigate whether:

- no outstanding asynchronous operation exists,
- the accept loop was never started,
- an exception aborted startup before operations were armed,
- the object owning the acceptor or timer was destroyed.

For long-running services, a work guard can help keep the event loop alive until controlled shutdown:

```
boost::asio::io_context ioc;

auto guard = boost::asio::make_work_guard(ioc);

std::thread worker([&]{
    ioc.run();
});

/* initialize async server here */

guard.reset();
worker.join();
```

Acceptor and connection lifecycle checklist

When the server is listening but clients cannot connect correctly, verify the accept path. Checklist:

- Is the acceptor bound to the expected address and port?
- Is the port already in use by another process?

- Is the firewall blocking the port?
- Is the program listening on `127.0.0.1` while the client connects to another interface?
- After each successful accept, is the next `async_accept` started?
- Is the accepted socket moved into a session object that remains alive?
- Does the session object inherit from `std::enable_shared_from_this` when needed?
- Are errors from accept ignored or logged?

A common bug is starting a session and forgetting to re-arm the accept loop. The debugging version below makes that obvious:

```
void do_accept()
{
    acceptor_.async_accept(
        [this](boost::system::error_code ec, tcp::socket socket)
        {
            if (ec) {
                std::cerr << "[ERR] async_accept: " << ec.message() << '\n';
            } else {
                std::cerr << "[DBG] accepted connection from "
                          << socket.remote_endpoint().address().to_string()
                          << ":" << socket.remote_endpoint().port() << '\n';

                std::make_shared<session>(std::move(socket))->start();
            }

            std::cerr << "[DBG] re-arming accept loop\n";
            do_accept();
```

```
        });
}
```

If the message `re-arming accept loop` is not reached, place a breakpoint inside the handler and inspect whether the code path exits early due to exception, conditional branch, or object destruction.

Session lifetime checklist

In asynchronous systems, many bugs are actually ownership bugs. The program appears to fail in networking code, but the real issue is that an object has already been destroyed.
Checklist:

- Does each active connection own the state required by pending operations?
- Are buffers, parsers, and responses alive until completion?
- Are lambdas capturing raw `this` where shared ownership is required?
- Are coroutine frame objects tied to a valid executor and socket lifetime?
- Is the response object in Beast kept alive until `async_write` completes?
- Are timers owned by the session rather than local stack variables that expire too soon?

Unsafe pattern:

```
void session::write_reply()
{
    http::response<http::string_body> res;
    res.result(http::status::ok);
    res.body() = "hello";
```

```
    res.prepare_payload();

    http::async_write(socket_, res,
        [this](boost::system::error_code ec, std::size_t)
        {
            if (ec) {
                std::cerr << ec.message() << '\n';
            }
        });
}
```

The response object above dies too early. Safer debugging-friendly pattern:

```
void session::write_reply()
{
    auto self = shared_from_this();

    auto res = std::make_shared<http::response<http::string_body>>();
    res->result(http::status::ok);
    res->set(http::field::server, "debug-server");
    res->set(http::field::content_type, "text/plain");
    res->body() = "hello";
    res->prepare_payload();

    http::async_write(socket_, *res,
        [self, res](boost::system::error_code ec, std::size_t bytes)
        {
            if (ec) {
                std::cerr << "[ERR] async_write: " << ec.message() << '\n';
                return;
```

```
            }

            std::cerr << "[DBG] async_write bytes = " << bytes << '\n';
        });
}
```

Whenever a bug looks random, first suspect ownership and lifetime.

Buffer and parser checklist

HTTP and TCP debugging often fails because the actual read/write boundary assumptions are wrong.
Checklist:

- Do not assume one read equals one complete message.
- Do not assume the entire request arrives in one packet.
- Do not reuse a buffer incorrectly across different protocol states.
- Confirm whether a persistent connection leaves unread bytes for the next request.
- For Beast, confirm parser limits and body handling rules.
- Check whether the client sent a request body larger than expected.

Instrument reads explicitly:

```
http::async_read(socket_, buffer_, request_,
    [self = shared_from_this()](boost::system::error_code ec, std::size_t bytes)
    {
        if (ec) {
```

```
            std::cerr << "[ERR] async_read: " << ec.message() << '\n';
            return;
        }

        std::cerr << "[DBG] request method  = " << self->request_.method_string()
        ↪  << '\n';
        std::cerr << "[DBG] request target  = " << self->request_.target() << '\n';
        std::cerr << "[DBG] request version = " << self->request_.version() << '\n';
        std::cerr << "[DBG] bytes read      = " << bytes << '\n';

        self->handle_request();
    });
```

When debugging malformed requests, log:

- method,
- target,
- version,
- content length,
- keep-alive state,
- parser/body size limits,
- remote endpoint.

Strand and executor checklist

Concurrency issues in `Boost.Asio` applications frequently arise when developers assume implicit serialization that does not actually exist.

Checklist:

- Does the session require serialized access to shared state?
- Are multiple handlers mutating the same object concurrently?
- If multiple threads call `run()`, should the session use a strand?
- Are timers, reads, writes, and shutdown paths all dispatched through the same executor?
- Are you mixing direct calls and posted calls in a way that breaks ordering assumptions?

A strand-based session skeleton:

```
class session : public std::enable_shared_from_this<session>
{
    tcp::socket socket_;
    boost::asio::strand<boost::asio::any_io_executor> strand_;

public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
        , strand_(socket_.get_executor())
    {
    }

    void start()
    {
        boost::asio::dispatch(strand_,
            [self = shared_from_this()]
            {
                self->do_read();
```

```
            });
    }

    void do_read()
    {
        /* all shared mutable state now stays serialized on strand_ */
    }
};
```

If a bug disappears when you force single-thread execution, you likely have an executor or synchronization problem rather than a protocol problem.

Timer and timeout checklist

Timeout bugs are among the most misunderstood in backend services. A request may fail due to cancellation caused by a timer, but the visible symptom is often reported as a read or write failure.
Checklist:

- Is the timeout timer started before the asynchronous operation?
- Is the timer cancelled when the operation completes successfully?
- Is the timeout callback racing with normal completion?
- Is timeout state logged with enough detail?
- Is a timed-out socket closed deliberately and consistently?

Useful timer instrumentation pattern:

```
void start_timeout()
{
    timer_.expires_after(std::chrono::seconds(30));

    timer_.async_wait(
        [self = shared_from_this()](boost::system::error_code ec)
        {
            if (ec == boost::asio::error::operation_aborted) {
                std::cerr << "[DBG] timer cancelled normally\n";
                return;
            }

            if (ec) {
                std::cerr << "[ERR] timer wait: " << ec.message() << '\n';
                return;
            }

            std::cerr << "[ERR] session timeout reached, closing socket\n";

            boost::system::error_code ignored;
            self->socket_.shutdown(tcp::socket::shutdown_both, ignored);
            self->socket_.close(ignored);
        });
}
```

Always distinguish between:

- a real timer expiration,
- a normal timer cancellation,
- another error in timer handling.

Error-code checklist

In Windows networking code, delayed or incomplete error logging is a serious debugging mistake. Error details must be captured immediately at the failing point.
Checklist:

- Never ignore `boost::system::error_code`.
- Log both numeric value and message text.
- In low-level Winsock paths, call `WSAGetLastError()` immediately after failure.
- Do not perform unrelated socket calls before retrieving the Winsock error.
- Distinguish transport failure, protocol failure, timeout, cancellation, and orderly shutdown.

A useful helper for Boost.System:

```
inline void log_ec(const char* where, const boost::system::error_code& ec)
{
    if (!ec) return;

    std::cerr << "[ERR] " << where
              << ": value=" << ec.value()
              << ", message=\"" << ec.message() << "\"\n";
}
```

For raw Winsock code on Windows:

```
#include <winsock2.h>
#include <windows.h>
```

```
#include <iostream>
#include <string>

std::string win_message(DWORD code)
{
    char* buffer = nullptr;

    DWORD n = ::FormatMessageA(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        nullptr,
        code,
        0,
        reinterpret_cast<LPSTR>(&buffer),
        0,
        nullptr);

    std::string msg = (n && buffer) ? buffer : "unknown error";
    if (buffer) {
        ::LocalFree(buffer);
    }
    return msg;
}

void report_winsock_failure(const char* where)
{
    int code = ::WSAGetLastError();
    std::cerr << "[ERR] " << where
```

```
              << ": WSAGetLastError=" << code
              << ", message=\"" << win_message(static_cast<DWORD>(code)) << "\"\n";
}
```

Use it immediately:

```
SOCKET s = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (s == INVALID_SOCKET) {
    report_winsock_failure("socket");
}
```

Exception checklist

In asynchronous servers, uncaught exceptions can terminate a worker thread, abort the session path, or cause silent failure if thrown in unexpected places.
Checklist:

- Are handler bodies wrapped in exception-safe boundaries where needed?
- Can parsing, routing, filesystem access, JSON handling, or user callbacks throw?
- Are thread entry points protected with top-level `try/catch`?
- In Visual Studio, are exception settings configured to break where useful?
- Are exceptions logged with sufficient request or connection context?

Worker-thread protection pattern:

```
std::thread t([&]{
    try {
        ioc.run();
```

```
    }
    catch (const std::exception& ex) {
        std::cerr << "[FATAL] exception from ioc.run(): " << ex.what() << '\n';
    }
    catch (...) {
        std::cerr << "[FATAL] unknown exception from ioc.run()\n";
    }
});
```

Handler protection pattern:

```
void session::handle_request()
{
    try {
        auto res = router_.dispatch(request_);
        send_response(std::move(res));
    }
    catch (const std::exception& ex) {
        std::cerr << "[ERR] request handling exception: " << ex.what() << '\n';
        send_internal_server_error();
    }
}
```

In Visual Studio, when chasing elusive faults, inspect exception settings and decide whether the debugger should break on thrown or unhandled exceptions for the relevant categories.

Visual Studio debugger checklist

Use the debugger as an execution-tracing system, not only as a crash catcher.
Checklist:

- Set breakpoints on startup, accept, read, write, timeout, and shutdown.
- Use the Call Stack window to verify how a handler was reached.
- Use Locals, Watch, and Autos to inspect parser state, socket state, and captured objects.
- Use conditional breakpoints for a specific request path, connection count, or error code.
- Use data breakpoints where applicable for state that changes unexpectedly.
- Inspect thread windows when running `io_context` on multiple threads.
- Toggle Just My Code appropriately when external frames hide important details.
- Use Natvis to improve readability of custom backend types in Watch windows.

An example of embedding Natvis for clearer debugging of custom state:

```
<?xml version="1.0" encoding="utf-8"?>
<AutoVisualizer xmlns="http://schemas.microsoft.com/vstudio/debugger/natvis/2010">
  <Type Name="server::request_context">
    <DisplayString>id={id}, method={method_}, path={path_}</DisplayString>
    <Expand>
      <Item Name="Request Id">id</Item>
      <Item Name="Method">method_</Item>
      <Item Name="Path">path_</Item>
      <Item Name="Remote Address">remote_addr_</Item>
    </Expand>
  </Type>
</AutoVisualizer>
```

When a type is difficult to inspect repeatedly during backend debugging, invest in Natvis early. It saves time across the entire project.

AddressSanitizer checklist

Memory corruption in backend services can appear as random socket failures, parser bugs, unexplained exceptions, or rare crashes under load. On Windows with MSVC, `/fsanitize=address` is one of the most valuable debugging tools for native C++ services. Checklist:

- Build a dedicated diagnostic configuration with `/fsanitize=address`.
- Reproduce the issue with the smallest realistic test case.
- Focus first on heap-buffer-overflow, use-after-free, double-free, and stack issues.
- Prefer deterministic repro cases over wide stress tests during first investigation.
- Keep debug symbols enabled.
- Do not assume a networking symptom implies a networking root cause; it may be memory corruption elsewhere.

Typical CMake diagnostic profile:

```
if (MSVC)
    add_compile_options(/fsanitize=address /Zi /Od /MDd /EHsc)
    add_link_options(/DEBUG:FULL)
endif()
```

A small ownership bug that ASan can expose:

```
#include <cstring>
#include <iostream>
#include <memory>
```

```
int main()
{
    auto p = std::make_unique<char[]>(8);
    std::strcpy(p.get(), "0123456789"); // overflow
    std::cout << p.get() << '\n';
}
```

In real backend software, a similar fault may appear through incorrectly sized buffers, unsafe string assembly, stale request storage, or invalid reuse of released memory.

Logging checklist

When a backend bug cannot be diagnosed through live stepping alone, structured logs become the primary source of truth.
Checklist:

- Assign a request ID or connection ID.
- Log accept, read, route, write, timeout, close, and error events.
- Include timestamps.
- Include remote endpoint information.
- Include bytes transferred where meaningful.
- Include the exact route or target being processed.
- Include whether the connection is keep-alive.
- Include executor thread identity if concurrency matters.

Minimal structured log helper:

```
#include <chrono>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <thread>

std::string now_text()
{
    using namespace std::chrono;
    auto t = system_clock::to_time_t(system_clock::now());

    std::tm tmv{};
    localtime_s(&tmv, &t);

    std::ostringstream oss;
    oss << std::put_time(&tmv, "%Y-%m-%d %H:%M:%S");
    return oss.str();
}

void log_line(std::string_view level,
              std::uint64_t id,
              std::string_view event,
              std::string_view detail)
{
    std::cerr << '[' << now_text() << "] "
              << '[' << level << "] "
              << "[conn=" << id << "] "
              << "[tid=" << std::this_thread::get_id() << "] "
              << event << " - " << detail << '\n';
```

```
}
```

Use it like this:

```
log_line("DBG", conn_id_, "accept", "session started");
log_line("DBG", conn_id_, "read", "waiting for request");
log_line("DBG", conn_id_, "route", "/api/status");
log_line("ERR", conn_id_, "write", ec.message());
```

HTTP routing checklist

When the server accepts connections correctly but still produces wrong responses, shift focus to routing and application logic.
Checklist:

- Is the request target normalized as expected?
- Are query strings handled intentionally?
- Are route comparisons case-sensitive or case-insensitive by design?
- Is the method checked before dispatch?
- Are unsupported methods returning a valid status code?
- Is body parsing performed only when appropriate?
- Is a moved-from request object accidentally reused?
- Is the response prepared with proper headers and payload length?

Simple route-debug pattern:

```
http::response<http::string_body>
route_request(const http::request<http::string_body>& req)
{
    std::cerr << "[DBG] route_request method="
              << req.method_string()
              << " target=" << req.target() << '\n';

    if (req.method() == http::verb::get && req.target() == "/health") {
        http::response<http::string_body> res{http::status::ok, req.version()};
        res.set(http::field::content_type, "text/plain");
        res.body() = "OK";
        res.prepare_payload();
        return res;
    }

    http::response<http::string_body> res{http::status::not_found, req.version()};
    res.set(http::field::content_type, "text/plain");
    res.body() = "Not Found";
    res.prepare_payload();
    return res;
}
```

When routes fail unexpectedly, print exactly what arrived instead of what was expected.

Graceful shutdown checklist

A server that cannot shut down cleanly is harder to debug because sockets, threads, and pending operations remain in uncertain states.
Checklist:

- Is shutdown initiated from a single clear control path?

- Are acceptors closed intentionally?
- Are timers cancelled?
- Are sockets shutdown before close where appropriate?
- Are worker threads joined?
- Are outstanding sessions allowed to complete or explicitly cancelled?
- Does process termination race against asynchronous cleanup?

A controlled shutdown sketch:

```
void server::stop()
{
    boost::system::error_code ec;

    acceptor_.cancel(ec);
    log_ec("acceptor.cancel", ec);

    acceptor_.close(ec);
    log_ec("acceptor.close", ec);

    ioc_.stop();
}
```

During shutdown debugging, differentiate between:

- normal cancellation,
- timeout-driven cancellation,

- peer-initiated close,
- application-requested shutdown,
- abnormal process termination.

Reproduction checklist

Many backend bugs become tractable only after a reliable reproduction strategy is defined. Checklist:

- Can the issue be reproduced with one client and one request?
- Does it require concurrency?
- Does it appear only with keep-alive connections?
- Does it depend on request size?
- Does it depend on timing or cancellation?
- Does it appear only in Release mode?
- Does it appear only when multiple threads call `run()`?
- Does it require TLS, file I/O, or external database interaction?

Create small reproducible clients when possible:

```
#include <boost/asio.hpp>
#include <iostream>

int main()
{
```

```
    try {
        boost::asio::io_context ioc;
        boost::asio::ip::tcp::resolver resolver{ioc};
        boost::asio::ip::tcp::socket socket{ioc};

        auto endpoints = resolver.resolve("127.0.0.1", "8080");
        boost::asio::connect(socket, endpoints);

        std::string req =
            "GET /health HTTP/1.1\r\n"
            "Host: 127.0.0.1\r\n"
            "Connection: close\r\n"
            "\r\n";

        boost::asio::write(socket, boost::asio::buffer(req));

        std::array<char, 4096> buf{};
        boost::system::error_code ec;
        std::size_t n = socket.read_some(boost::asio::buffer(buf), ec);

        std::cout.write(buf.data(), static_cast<std::streamsize>(n));
        std::cout << '\n';

        if (ec && ec != boost::asio::error::eof) {
            std::cerr << ec.message() << '\n';
        }
    }
    catch (const std::exception& ex) {
        std::cerr << ex.what() << '\n';
```

```
    }
}
```

A small deterministic client is often more useful than a complex external testing tool during initial debugging.

Pre-release debugging checklist

Before considering the backend stable, verify the following under Windows:

- debug build runs without unexpected exceptions,
- sanitizer build runs without memory errors,
- route handling is verified for success and failure paths,
- invalid requests produce correct error responses,
- idle timeout behaves correctly,
- shutdown path is clean,
- no request path relies on undefined lifetime,
- logging is sufficient to reconstruct a failing request,
- multiple concurrent connections do not corrupt shared state,
- socket and timer errors are always logged with context.

Final field checklist

When a real defect appears, follow this order:

1. Reproduce the issue in the smallest possible Windows debug build.
2. Confirm startup, bind, listen, and event-loop execution.
3. Trace accept, read, route, write, and shutdown path.
4. Inspect ownership and lifetime of all state captured by async operations.
5. Log every error code immediately.
6. Check timer cancellation and timeout races.
7. Check strand or executor serialization assumptions.
8. Use Visual Studio exception settings and call stacks deliberately.
9. Rebuild with AddressSanitizer when corruption is plausible.
10. Fix the first verified root cause, then rerun the full checklist.

The discipline of backend debugging is not in learning one tool, but in enforcing a repeatable investigation model. `Boost.Asio` and `Boost.Beast` provide a powerful and correct asynchronous foundation, but correctness in a production backend depends on the engineer's ability to reason about execution order, state ownership, cancellation, protocol boundaries, and platform-level diagnostics. A strong debugging checklist turns that reasoning into a professional engineering habit.

Appendix C — Performance Tips

This appendix presents a practical performance guide for building high-quality backend services with `Boost.Asio` and `Boost.Beast` on Windows. The emphasis is on techniques that align with the official asynchronous execution model, executor and strand usage, buffer management rules, and modern MSVC performance tooling. The goal is not to chase micro-optimizations blindly, but to help the reader build systems whose throughput, latency, scalability, and stability improve together.

In backend engineering, performance is not one number. A service may have excellent raw throughput and still feel slow because tail latency is poor. Another service may respond quickly for one client but collapse under load because it creates too many threads, reallocates buffers excessively, copies messages unnecessarily, or serializes unrelated work through a single lock. For this reason, the most effective performance work begins with architecture and execution flow rather than with isolated instruction-level tuning.

Performance mindset for asynchronous backends

Before tuning specific code paths, keep the following principles in mind:

- Prefer reducing unnecessary work over making unnecessary work faster.
- Prefer stable latency over peak benchmark numbers that collapse under load.
- Prefer a clear execution model over accidental concurrency.
- Prefer measuring real hot paths over guessing.
- Prefer fewer copies, fewer allocations, fewer context switches, and fewer locks.
- Prefer protocol-aware buffering over ad hoc string assembly.

In `Boost.Asio`-based systems, the largest performance gains usually come from a short list of core decisions:

- how many threads execute `io_context::run()`,
- whether a strand is used correctly,
- whether session lifetime is efficient,
- whether requests and responses are copied too often,
- whether buffers are reused,
- whether slow work blocks I/O paths,
- whether the build configuration is optimized properly on Windows,
- and whether profiling confirms the true bottleneck.

Choose the right concurrency model

One of the most common mistakes in backend code is assuming that “more threads” always means “more performance”. In practice, too many threads can degrade performance through scheduler overhead, cache disruption, lock contention, and increased context switching.
A good baseline design on Windows is:

- one `io_context`,
- a fixed number of worker threads,
- asynchronous socket operations,
- per-session serialization only where needed,

- and no blocking work on the I/O threads.

A common starting point is to size the worker pool near the number of logical CPU cores, then measure under realistic traffic. For many services, this is far better than creating one thread per connection.

```
#include <boost/asio.hpp>
#include <iostream>
#include <thread>
#include <vector>

namespace asio = boost::asio;

int main()
{
    asio::io_context ioc;

    const unsigned thread_count =
        std::max(1u, std::thread::hardware_concurrency());

    auto guard = asio::make_work_guard(ioc);

    std::vector<std::thread> workers;
    workers.reserve(thread_count);

    for (unsigned i = 0; i < thread_count; ++i) {
        workers.emplace_back([&ioc, i] {
            try {
                ioc.run();
            } catch (const std::exception& ex) {
```

```
                std::cerr << "worker " << i
                          << " exception: " << ex.what() << '\n';
            }
        });
    }

    std::cout << "I/O threads running: " << thread_count << '\n';

    guard.reset();

    for (auto& t : workers) {
        t.join();
    }
}
```

This does not mean that the ideal thread count is always equal to hardware concurrency. It means that performance tuning should begin from a controlled fixed-size pool, not from unbounded thread creation.

Use strands only where serialization is required

A strand provides sequential invocation of handlers and is one of the cleanest ways to avoid explicit locking in shared asynchronous state. It is extremely useful, but it is also possible to overuse it and accidentally serialize unrelated work.
Use a strand when:

- multiple handlers access the same mutable session state,
- reads, writes, timers, and shutdown logic can race,

- or you want one session to be logically single-threaded even when the `io_context` runs on many threads.

Avoid using one global strand for all sessions. That would turn a scalable server into a single serialized pipeline.
Efficient per-session pattern:

```
#include <boost/asio.hpp>
#include <memory>

namespace asio = boost::asio;
using tcp = asio::ip::tcp;

class session : public std::enable_shared_from_this<session>
{
    tcp::socket socket_;
    asio::strand<asio::any_io_executor> strand_;

public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
        , strand_(socket_.get_executor())
    {
    }

    void start()
    {
        asio::dispatch(strand_,
            [self = shared_from_this()]
            {
```

```
                self->do_read();
            });
    }

private:
    void do_read()
    {
    }
};
```

Performance rule:

> Serialize only the state that truly requires serialization.

That keeps correctness while preserving parallelism across independent sessions.

Do not block I/O threads

A backend built on asynchronous I/O loses much of its value if handlers perform long blocking tasks such as:

- database calls through blocking APIs,
- filesystem scans,
- compression of large payloads,
- CPU-heavy JSON transformation,
- image processing,
- or long business-logic loops.

I/O threads should initiate, coordinate, and complete network work. Expensive tasks should be offloaded to a separate execution context or worker pool.
A simple split between network I/O and background compute:

```
#include <boost/asio.hpp>
#include <iostream>
#include <thread>
#include <vector>

namespace asio = boost::asio;

int main()
{
    asio::io_context net_ioc;
    asio::thread_pool cpu_pool(4);

    asio::post(net_ioc, [&] {
        asio::post(cpu_pool, [] {
            /* expensive CPU-bound task */
        });
    });

    std::thread net_thread([&] { net_ioc.run(); });

    net_ioc.stop();
    net_thread.join();
    cpu_pool.join();
}
```

A service can appear fully asynchronous on the surface and still perform poorly because

expensive work is being done on the wrong executor.

Minimize dynamic allocations in hot paths

Allocations are not always expensive individually, but repeated allocations in high-frequency paths often become a measurable bottleneck. In HTTP services, common allocation sources include:

- building large response strings repeatedly,
- constructing temporary containers per request,
- repeatedly allocating parser state,
- repeatedly growing buffers,
- and creating short-lived heap objects for every minor operation.

Useful strategies:

- reuse session-owned buffers,
- reserve capacity when sizes are predictable,
- prefer `std::string::reserve()` before repeated appends,
- keep request parsing storage local to the connection lifecycle,
- avoid converting between multiple intermediate string forms.

Example: reserve response storage before assembly.

```cpp
#include <string>

std::string make_json_status()
{
    std::string body;
    body.reserve(256);

    body += "{";
    body += "\"service\":\"backend\",";
    body += "\"status\":\"ok\",";
    body += "\"version\":\"1.0\"";
    body += "}";

    return body;
}
```

This pattern is simple, portable, and often more efficient than repeated unpredictable growth.

Reuse buffers deliberately

Buffer management is one of the most important performance topics in real Asio and Beast systems. The official buffer model is designed around buffer sequences and dynamic buffers. In practice, this means you should avoid re-creating input storage unnecessarily when the same connection will handle repeated traffic.
For HTTP session objects, a common pattern is:

- store one `beast::flat_buffer` per session,
- reuse it across request reads,
- consume or clear it correctly between message boundaries,

- avoid copying incoming data into multiple extra containers.

Example session state:

```
#include <boost/beast/core/flat_buffer.hpp>
#include <boost/beast/http.hpp>
#include <boost/asio.hpp>
#include <memory>

namespace beast = boost::beast;
namespace http  = beast::http;
namespace asio  = boost::asio;
using tcp = asio::ip::tcp;

class session : public std::enable_shared_from_this<session>
{
    tcp::socket socket_;
    beast::flat_buffer buffer_;
    http::request<http::string_body> req_;

public:
    explicit session(tcp::socket socket)
        : socket_(std::move(socket))
    {
    }

    void start()
    {
        do_read();
    }
```

```
private:
    void do_read()
    {
        req_ = {};

        http::async_read(socket_, buffer_, req_,
            [self = shared_from_this()](beast::error_code ec, std::size_t bytes)
            {
                if (ec) {
                    return;
                }

                self->handle_request();
            });
    }

    void handle_request()
    {
    }
};
```

The important point is not merely that `flat_buffer` exists, but that reusing a connection-owned buffer reduces repeated allocation churn and preserves a clear ownership model.

Avoid unnecessary copies of requests and responses

Copying large HTTP messages or bodies can become expensive under load. Whenever possible:

- pass requests by reference,
- move responses when ownership transfer is appropriate,
- avoid serializing through extra temporary strings,
- and keep body construction close to the final response object.

A clean request dispatcher signature:

```
#include <boost/beast/http.hpp>
#include <string>

namespace http = boost::beast::http;

http::response<http::string_body>
route_request(const http::request<http::string_body>& req)
{
    http::response<http::string_body> res{http::status::ok, req.version()};
    res.set(http::field::content_type, "text/plain");
    res.body() = "Hello from Boost.Beast";
    res.prepare_payload();
    return res;
}
```

Do not create extra copies of `req.body()` or whole message objects unless the logic truly needs them.

Prefer persistent connections when appropriate

For HTTP/1.1 services, repeatedly reconnecting for every request increases cost through:

- more socket creation,
- more kernel work,
- more TCP handshaking,
- more allocator churn,
- and more request startup overhead.

If the service design and client behavior permit it, keep-alive can improve overall throughput and reduce per-request latency.
A simple response that preserves keep-alive state:

```
http::response<http::string_body>
make_response(const http::request<http::string_body>& req, std::string body)
{
    http::response<http::string_body> res{http::status::ok, req.version()};
    res.set(http::field::server, "asio-backend");
    res.set(http::field::content_type, "text/plain");
    res.keep_alive(req.keep_alive());
    res.body() = std::move(body);
    res.prepare_payload();
    return res;
}
```

However, persistent connections also require disciplined timeout handling and proper parser state management. Performance improves only when lifecycle correctness is maintained.

Set parser and message limits consciously

A performance-friendly server is not merely fast for valid traffic. It also avoids excessive resource consumption on hostile or malformed traffic. Unbounded parsing can lead to memory pressure, unnecessary allocations, and degraded throughput.
Practical policy examples include:

- limit accepted body size,
- reject oversized headers,
- apply timeouts for slow clients,
- and avoid reading arbitrarily large messages into memory without a reason.

Illustrative parser setup:

```
#include <boost/beast/http.hpp>

namespace http = boost::beast::http;

void configure_parser(http::request_parser<http::string_body>& parser)
{
    parser.body_limit(1024 * 1024); // 1 MiB
}
```

This is both a safety measure and a performance measure. It keeps resource usage predictable.

Batch work where it is sensible

Not every optimization must be sophisticated. Sometimes performance improves because multiple tiny operations are combined into one larger efficient operation.
Examples:

- reserve output strings before multiple appends,
- aggregate log output,
- send complete HTTP responses rather than fragmented custom writes,
- and avoid posting many tiny tasks when one larger task preserves the same semantics.

Poor pattern:

```
asio::post(ioc, []{ /* step 1 */ });
asio::post(ioc, []{ /* step 2 */ });
asio::post(ioc, []{ /* step 3 */ });
```

Often better:

```
asio::post(ioc, []{
    /* step 1 */
    /* step 2 */
    /* step 3 */
});
```

This is not a universal rule. Sometimes splitting work is necessary for fairness or cancellation. But excessive micro-posting can create avoidable scheduler overhead.

Use coroutines to simplify control flow, not to hide cost

Modern Boost.Asio supports coroutine-oriented asynchronous code. Coroutines often improve maintainability by making control flow clearer than deeply nested callbacks. That clarity can also help performance work, because it becomes easier to reason about actual operation order, cancellation paths, and lifetime.
But coroutine syntax does not automatically make code faster. Performance still depends on:

- allocation behavior,
- executor choice,
- copies,
- blocking calls,
- and protocol design.

Simple coroutine-style outline:

```
#include <boost/asio.hpp>

namespace asio = boost::asio;

asio::awaitable<void> do_work()
{
    co_return;
}
```

Use coroutine style when it makes the session flow clearer and easier to maintain. That can indirectly improve performance because the code becomes easier to profile and optimize correctly.

Reduce locking and shared-state contention

Locks are sometimes necessary, but shared mutable global state often becomes a scalability bottleneck in backend systems.
Common sources of contention:

- one global map updated on every request,

- one mutex around a central logger,
- one shared statistics object with frequent writes,
- one cache protected by coarse-grained locking,
- one queue used by all sessions.

Strategies:

- keep state per session when possible,
- shard shared structures,
- collect counters per thread and merge periodically,
- use strands for per-object serialization,
- reduce critical-section duration aggressively.

Illustrative sharded counter design:

```
#include <array>
#include <atomic>
#include <cstddef>

struct counters
{
    static constexpr std::size_t shards = 16;
    std::array<std::atomic<std::uint64_t>, shards> requests{};

    void add(std::size_t shard_index)
    {
```

```
        requests[shard_index % shards].fetch_add(1, std::memory_order_relaxed);
    }

    std::uint64_t total() const
    {
        std::uint64_t sum = 0;
        for (const auto& c : requests) {
            sum += c.load(std::memory_order_relaxed);
        }
        return sum;
    }
};
```

This is often more scalable than one mutex-guarded counter under very high concurrency.

Keep response bodies cheap when possible

Dynamic response generation is useful, but not every response needs expensive assembly. For hot endpoints such as:

- `/health`,
- `/ready`,
- static configuration snapshots,
- simple status responses,
- or small routing metadata,

prefer compact cheap bodies.
Example:

```
http::response<http::string_body>
make_health_response(unsigned version, bool keep_alive)
{
    http::response<http::string_body> res{http::status::ok, version};
    res.set(http::field::content_type, "text/plain");
    res.keep_alive(keep_alive);
    res.body() = "OK";
    res.prepare_payload();
    return res;
}
```

For very hot paths, even small simplifications in body generation and header setup can matter when multiplied across millions of requests.

Serve files efficiently and avoid wasteful file handling

Static file serving on Windows can suffer if the code repeatedly:

- opens files unnecessarily,
- copies file contents into extra strings,
- computes metadata repeatedly,
- or performs synchronous disk work on threads that should stay focused on networking.

General rules:

- normalize document-root handling once,
- avoid repeated path reconstruction when possible,
- cache small static assets if they are requested frequently and fit memory budgets,

- be careful with synchronous file access inside high-frequency request handlers.

Simple path assembly with pre-reserved storage:

```
#include <string>
#include <string_view>

std::string build_path(std::string_view doc_root, std::string_view target)
{
    std::string path;
    path.reserve(doc_root.size() + target.size() + 8);
    path.append(doc_root.data(), doc_root.size());

    if (!path.empty() && path.back() == '\\' && !target.empty() && target.front()
    ↪  == '/') {
        path.pop_back();
    }

    path.append(target.data(), target.size());
    return path;
}
```

The exact file-serving strategy depends on workload, but path churn and repeated allocations should still be minimized.

Use the correct MSVC optimization settings for Release builds

On Windows, substantial performance gains often come from correct compiler and linker settings before any source-level change is made.
For normal Release-style native builds, common goals include:

- full optimization,
- function inlining,
- whole-program analysis when appropriate,
- link-time code generation,
- and architecture-appropriate code generation.

Typical MSVC-oriented Release guidance:

```
C/C++ > Optimization > Optimization: Maximize Speed (/O2)
C/C++ > Optimization > Intrinsic Functions: Yes (/Oi)
C/C++ > Optimization > Favor Size or Speed: Favor fast code (/Ot)
C/C++ > General > Whole Program Optimization: Yes (/GL)
Linker > Optimization > Link Time Code Generation: Use Link Time Code Generation
↪  (/LTCG)
```

A practical CMake setup on Windows:

```
cmake_minimum_required(VERSION 3.20)
project(backend LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(backend main.cpp)

if (MSVC)
    target_compile_options(backend PRIVATE
        /O2
```

```
        /Oi
        /Ot
        /EHsc
    )

    target_link_options(backend PRIVATE
        /LTCG
    )
endif()
```

Do not confuse debugging builds with final performance builds. Measure both. Debug builds are for correctness investigation; Release builds are for realistic performance measurement.

Use profile-guided optimization for stable hot paths

When the codebase and workload are stable enough, PGO can produce meaningful gains by teaching the optimizer which paths are actually hot.
PGO is especially useful when:

- the service has clear dominant routes,
- branch behavior is repetitive,
- or the codebase is large enough that whole-program layout and inlining choices matter.

Conceptually, the workflow is:

1. build an instrumented binary,
2. run representative workload,
3. rebuild using the collected profile data.

PGO is only worthwhile when the training workload is realistic. A poor training workload can optimize the wrong behavior.

Measure with the Visual Studio profiler before guessing

Many performance efforts fail because the wrong bottleneck is targeted. Windows developers should use the Visual Studio profiling tools deliberately.
Useful profiler categories include:

- CPU usage,
- instrumentation,
- memory allocation views,
- and build-specific optimized-code analysis.

A disciplined workflow:

1. build a Release configuration with symbols,
2. run a realistic test workload,
3. capture CPU hotspots,
4. inspect self time and total time,
5. check whether the bottleneck is in networking, parsing, string construction, synchronization, logging, or application logic.

Only after the hotspot is confirmed should code be changed.

Debug optimized builds when the bug only appears in Release

Some performance-sensitive bugs appear only under optimization. On current Microsoft tooling, optimized-code debugging support is significantly better than in older toolchains. This is important because backend services often behave differently in Release under real scheduling and inlining.
Practical advice:

- keep a special diagnostic Release configuration with symbols,
- enable optimization while retaining debuggability,
- reproduce the issue there,
- compare against Debug only after the optimized behavior is understood.

Example configuration concept:

```
if (MSVC)
    target_compile_options(backend PRIVATE
        $<$<CONFIG:RelWithDebInfo>:/O2>
        $<$<CONFIG:RelWithDebInfo>:/Zi>
    )
    target_link_options(backend PRIVATE
        $<$<CONFIG:RelWithDebInfo>:/DEBUG:FULL>
    )
endif()
```

This is often the right compromise when correctness and performance must be inspected together.

Be careful with logging on hot paths

Logging is essential for observability, but heavy synchronous logging can easily become a performance problem.
Expensive logging patterns include:

- formatting large strings for every request,
- flushing output streams too often,
- writing synchronously from many threads,
- and logging full headers or bodies in high-volume production paths.

Practical strategies:

- log summaries instead of full payloads on hot paths,
- batch or queue logs when practical,
- use log levels aggressively,
- disable verbose diagnostic logging in throughput benchmarks.

Simple level-guarded logging:

```
#include <iostream>
#include <string_view>

enum class log_level
{
    error,
    info,
```

```
    debug
};

log_level current_level = log_level::info;

void log_message(log_level level, std::string_view text)
{
    if (static_cast<int>(level) > static_cast<int>(current_level)) {
        return;
    }

    std::cerr << text << '\n';
}
```

A benchmark that includes full debug logging is often benchmarking the logger more than the backend itself.

Use timeouts to protect performance

Timeouts are not only for correctness. They are also performance controls. Slow clients can consume:

- connection slots,
- buffer space,
- session objects,
- thread attention,
- and overall service capacity.

Reasonable read, write, and idle timeouts help preserve the performance envelope of the service under poor client behavior.

Illustrative timeout setup:

```
#include <boost/asio.hpp>
#include <chrono>

namespace asio = boost::asio;

class timeout_guard
{
    asio::steady_timer timer_;

public:
    explicit timeout_guard(asio::any_io_executor ex)
        : timer_(ex)
    {
    }

    void start()
    {
        timer_.expires_after(std::chrono::seconds(30));
        timer_.async_wait([](const boost::system::error_code& ec) {
            if (!ec) {
                /* handle timeout */
            }
        });
    }

    void cancel()
```

```
    {
        timer_.cancel();
    }
};
```

A backend that never releases slow or stalled connections will lose performance even if its fast-path code is excellent.

Prefer stable session objects over fragmented state

Performance often improves when a session owns its working state directly rather than scattering it across many tiny heap allocations.
A productive per-connection layout usually contains:

- the socket,
- the strand if needed,
- the input buffer,
- the current request,
- the response or write state,
- the timer,
- and lightweight route-local scratch storage.

This improves locality, simplifies ownership, and reduces accidental lifetime bugs that can also hurt performance.

Benchmark with realistic traffic patterns

Many backend microbenchmarks are misleading because they test unrealistic cases such as:

- a single route only,
- tiny requests only,
- localhost only,
- no keep-alive,
- no concurrency,
- no logging,
- or no malformed traffic.

Useful benchmark dimensions include:

- concurrent clients,
- small and medium payloads,
- keep-alive versus reconnect-per-request,
- route mix,
- success and failure cases,
- timeout pressure,
- warm cache and cold cache behavior.

A strong performance result should answer not only “how fast?” but also “under what traffic pattern, with what latency distribution, using what build configuration?”

Check the allocator and container behavior of your design

Even when the networking layer is sound, application-level containers can dominate runtime costs. Watch for:

- maps rebuilt on every request,
- vectors that grow without reserve,
- string-heavy routing tables,
- repeated conversion between UTF forms,
- JSON DOM creation when a streaming approach would suffice.

Example of one-time route table initialization:

```
#include <string>
#include <unordered_map>
#include <functional>

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

std::unordered_map<std::string, handler_fn> build_routes()
{
    std::unordered_map<std::string, handler_fn> routes;
    routes.reserve(32);

    routes.emplace("/health", []{});
    routes.emplace("/status", []{});
    routes.emplace("/metrics", []{});
```

```
    return routes;
}
```

Initialize stable structures once when possible. Rebuilding them per request is needless work.

Exploit cheap protocol paths

A mature backend usually has a small number of extremely common operations. These should have direct, efficient code paths.
Examples:

- a direct `/health` response,
- a precomputed status payload,
- an early method check,
- a fast not-found path,
- skipping unnecessary body parsing for `GET` requests.

Fast early method validation:

```
http::response<http::string_body>
dispatch(const http::request<http::string_body>& req)
{
    if (req.method() != http::verb::get &&
        req.method() != http::verb::post)
    {
        http::response<http::string_body> res{
            http::status::method_not_allowed, req.version()
        };
```

```
        res.set(http::field::content_type, "text/plain");
        res.body() = "Method Not Allowed";
        res.prepare_payload();
        return res;
    }

    return route_request(req);
}
```

Cheap rejection of unsupported traffic prevents wasted downstream work.

Use AddressSanitizer during tuning to avoid false wins

Performance work can accidentally introduce memory corruption, lifetime errors, or out-of-bounds access that seem harmless in small tests but collapse under real traffic. On Windows, MSVC AddressSanitizer remains an important validation tool.
Use it when:

- refactoring buffers,
- changing ownership,
- introducing custom memory handling,
- or modifying hot paths in ways that remove safety checks.

Simple CMake diagnostic profile:

```
if (MSVC)
    target_compile_options(backend PRIVATE
        $<$<CONFIG:AsanDebug>:/fsanitize=address>
        $<$<CONFIG:AsanDebug>:/Zi>
```

```
        $<$<CONFIG:AsanDebug>:/Od>
    )
    target_link_options(backend PRIVATE
        $<$<CONFIG:AsanDebug>:/DEBUG:FULL>
    )
endif()
```

A performance improvement is not real if it relies on undefined behavior.

Keep Windows-specific deployment realities in mind

Performance on Windows is influenced not only by code, but also by deployment choices such as:

- antivirus interaction with binaries and logs,
- firewall configuration,
- certificate configuration for TLS,
- power plan and CPU frequency policy,
- background tools attached during measurement,
- and whether the benchmark runs on localhost or across a real network path.

For serious measurements:

- use a clean Release build,
- keep symbols if profiling,
- disable unnecessary diagnostic noise,

- document the machine, compiler, Boost version, and workload shape,
- and run repeated tests rather than relying on a single number.

A compact practical checklist

Use the following checklist before deep low-level tuning:

1. Confirm that no blocking work runs on I/O threads.
2. Use a fixed worker pool, not thread-per-connection.
3. Apply strands only to state that truly needs serialization.
4. Reuse buffers and avoid repeated allocation churn.
5. Avoid unnecessary request and response copies.
6. Limit body sizes and protect the parser.
7. Use keep-alive where appropriate.
8. Reduce coarse-grained locking and shared contention.
9. Build and measure true Release configurations on Windows.
10. Use profiling before changing source code.
11. Use PGO when workloads are stable and representative.
12. Validate tuning changes with AddressSanitizer and realistic traffic.

Final guidance

The most important performance lesson in asynchronous backend development is that architecture dominates micro-optimization. A backend based on `Boost.Asio` can scale impressively on Windows when its execution model is disciplined, its buffers are reused intelligently, its session state is structured clearly, and its Release build is measured with the proper Microsoft tooling.

In practice, the biggest wins usually come from a few disciplined habits: keeping I/O threads non-blocking, minimizing allocations and copies, avoiding unnecessary serialization, and profiling the actual workload before tuning. Once those habits are established, the rest of performance engineering becomes much more precise, much more repeatable, and much more valuable.

References

Boost.Asio official documentation

Scope and role of the official Asio documentation

The official `Boost.Asio` documentation is the primary reference for the execution model used throughout this book. In current Boost documentation, `Asio` is presented not merely as a socket wrapper, but as a complete asynchronous I/O framework built around execution contexts, executors, completion tokens, cancellation, buffers, timers, networking objects, and coroutine support. For backend engineering on Windows, this documentation is the authoritative source for understanding how to structure an event-driven server that remains correct under concurrency and scalable under load.

In practice, the official `Asio` documentation should be read in four layers:

1. the overview pages, which explain the design model,
2. the core concepts pages, which define the rules for handlers, executors, strands, buffers, and operation completion,
3. the reference pages, which define exact APIs and signatures,
4. and the revision history, which documents important correctness and behavior changes across Boost releases.

For serious Windows backend development, all four layers matter. The overview explains what the library is trying to guarantee. The core concepts explain how those guarantees are achieved. The reference pages define the exact contract the code must follow. The release history shows where important behavior around cancellation, coroutine support, executor semantics, or platform integration has evolved over time.

The official conceptual structure to study first

The official `Asio` overview should be the first stop for any reader who wants to master backend design with `Boost.Asio`. Its most important areas for server engineering are these:

- asynchronous operations,
- asynchronous agents,
- executors,
- allocators,
- cancellation,
- completion tokens,
- strands,
- buffers,
- streams with short reads and short writes,
- C++20 coroutine support,
- Windows-specific functionality.

This matters because many backend problems come not from syntax or networking APIs, but from misunderstanding these structural rules. A server that compiles successfully may still fail at runtime if these concepts are not applied correctly.

Execution model and io_context

The execution model described in the official documentation is centered around `io_context`. This object represents the event loop responsible for dispatching completion handlers.

```
#include <boost/asio.hpp>
#include <iostream>

namespace asio = boost::asio;

int main()
{
    asio::io_context ioc;

    asio::post(ioc, [] {
        std::cout << "Hello from io_context\n";
    });

    ioc.run();
}
```

The official documentation emphasizes that asynchronous operations do not complete immediately. Instead, they initiate work and later deliver results through handlers, coroutines, or other completion tokens. This separation is fundamental for scalability.

Completion tokens and async model

Modern `Boost.Asio` relies heavily on completion tokens. These allow the same asynchronous function to support multiple styles:

- callback-based,
- future-based,
- coroutine-based.

Example with callback:

```
socket.async_read_some(buffer,
    [](boost::system::error_code ec, std::size_t bytes)
    {
        if (!ec) {
        }
    });
```

Example with coroutine:

```
#include <boost/asio.hpp>

namespace asio = boost::asio;

asio::awaitable<void> read_task(asio::ip::tcp::socket socket)
{
    char data[1024];

    std::size_t n = co_await socket.async_read_some(
        asio::buffer(data),
```

```
        asio::use_awaitable);

    (void)n;
}
```

The official documentation highlights that completion tokens unify asynchronous programming styles without changing the underlying operation.

Boost.Beast official documentation

Role of Beast in backend development

`Boost.Beast` is built on top of `Boost.Asio` and provides HTTP and WebSocket functionality. The official documentation focuses on message-oriented networking, where parsing and serialization follow strict protocol rules.
For backend engineering, Beast is the standard approach to implementing HTTP servers and clients using Asio's asynchronous model.

HTTP request and response model

Beast defines HTTP messages as strongly typed structures.

```
#include <boost/beast/http.hpp>

namespace http = boost::beast::http;

http::response<http::string_body>
make_response(unsigned version)
{
    http::response<http::string_body> res{http::status::ok, version};
```

```
    res.set(http::field::content_type, "text/plain");
    res.body() = "Hello from Beast";
    res.prepare_payload();
    return res;
}
```

The official documentation emphasizes correct header handling, payload preparation, and protocol compliance.

Asynchronous HTTP server pattern

A canonical Beast server follows this lifecycle:

- accept connection,
- read request,
- process request,
- write response,
- optionally keep connection alive.

```
http::async_read(socket, buffer, request,
    [self](boost::system::error_code ec, std::size_t)
    {
        if (!ec) {
            self->handle_request();
        }
    });
```

The documentation stresses that all operations must respect buffer ownership and lifetime rules.

ISO C++ (C++20/23 coroutines)

Coroutine model in ISO C++

C++20 introduced language-level coroutines, which provide a structured way to express asynchronous workflows without deeply nested callbacks. The official ISO model defines coroutines through:

- promise types,
- coroutine handles,
- suspension points,
- awaitable objects.

These form the foundation used by `Boost.Asio` for coroutine-based networking.

Basic coroutine example

```
#include <coroutine>
#include <iostream>

struct simple_task
{
    struct promise_type
    {
        simple_task get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
```

```
        void unhandled_exception() {}
    };
};

simple_task example()
{
    std::cout << "Coroutine executed\n";
    co_return;
}
```

This example demonstrates the structure defined by the standard.

Coroutines with Boost.Asio

In real backend systems, coroutines integrate with `Boost.Asio` using `awaitable` and `co_await`.

```
#include <boost/asio.hpp>

namespace asio = boost::asio;

asio::awaitable<void> session(asio::ip::tcp::socket socket)
{
    char data[1024];

    std::size_t n = co_await socket.async_read_some(
        asio::buffer(data),
        asio::use_awaitable);

    (void)n;
```

```
    co_return;
}
```

The combination of ISO coroutines and Asio executors forms the modern foundation for writing clean, scalable backend systems in C++.

Final perspective

The combination of `Boost.Asio`, `Boost.Beast`, and ISO C++ coroutines represents a mature and powerful backend stack. The official documentation for these technologies provides not only API references, but also a complete conceptual model for building high-performance asynchronous systems on Windows.
Mastering these sources enables the developer to move beyond simple networking code toward designing full backend architectures that are efficient, scalable, and maintainable.