

FrameWorks

Servicios Web C++

Antonio Espín Herranz

Contenidos

- **Boost.Beast:** Manejo de HTTP y WebSockets en C++.
- Librería **crow**
- Comparación y selección de herramientas según el tipo de aplicación

Boost.Beast

Boost.Beast

- Nos proporciona manejo de peticiones **Http** y **WebSockets**
- Para **instalar**:
 - **vcpkg install boost-beast**
- **Integrar** en Visual Studio:
 - **vcpkg integrate install**
- Listar librerías:
 - **vcpkg list**
- Comprobar si la tenemos instalada:
 - **ccpkg list | findstr boost-beast**

Características de la librería

- Boost.Beast es una biblioteca **header-only** (solo incluye cabeceras, no requiere compilación previa) que proporciona componentes de bajo nivel para manejar protocolos de red como **HTTP/1** y **WebSocket**,
- Con soporte para operaciones **síncronas** y **asíncronas**

Características de la librería II

- Basada en **Boost.Asio**
 - Usa el modelo asincrónico de Boost.Asio, lo que permite construir aplicaciones altamente concurrentes.
 - Compatible con `io_context`, `executors`, y operaciones compuestas.
- Protocolos soportados
 - **HTTP/1.1**: Lectura, escritura, serialización y análisis de mensajes HTTP.
 - **WebSocket**: Comunicación bidireccional en tiempo real, incluyendo control frames y compresión (permessage-deflate).

Características III

- Abstracciones de flujo (Streams)
 - `basic_stream`, `tcp_stream`, `ssl_stream`: para manejar conexiones TCP/IP, con o sin cifrado.
- Soporte para SSL/TLS mediante integración con OpenSSL.
- Gestión de buffers
 - `flat_buffer`, `multi_buffer`, `static_buffer`: para optimizar el manejo de datos en red.
- Flexibilidad
 - El desarrollador controla aspectos como el manejo de buffers, hilos, y políticas de tasa de transferencia.
 - Ideal para construir tanto clientes como servidores, gracias a su diseño simétrico.
- Extensibilidad
 - Sirve como base para construir bibliotecas de red más complejas.
 - Bien adaptada para integrarse en arquitecturas de microservicios o sistemas distribuidos.

Requisitos

- Página oficial: <https://www.boost.org/library/latest/beast/>
- Wiki: <https://deepwiki.com/boostorg/beast>
- Necesitamos versiones **\geq C++11**
- **Boost.Asio** y otras partes de Boost.
- **OpenSSL** si se desea soporte para conexiones seguras.
- Compatible con Visual Studio 2017+, CMake \geq 3.5.1, para construir ejemplos

Desglose de la librería Boost.Beast

Núcleo de Boost.Beast

- **Buffers y Streams:** Beast proporciona sus propios tipos de buffer (`flat_buffer`) y abstrae los streams para facilitar la lectura/escritura de datos en conexiones TCP.
- **Integración con Boost.Asio:** Toda la funcionalidad de Beast se basa en Asio, lo que permite usar operaciones sincrónicas y asincrónicas con coroutines, callbacks o `async/await`.

Manejo de HTTP

- **HTTP Messages**

- `http::request<T>` y `http::response<T>`: Representan mensajes HTTP con cuerpo de tipo T (como `string_body`, `file_body`, etc.).
- `http::fields`: Encapsula los encabezados HTTP.

- **HTTP Operations**

- `http::read` / `http::async_read`: Leer peticiones o respuestas desde un stream.
- `http::write` / `http::async_write`: Enviar peticiones o respuestas por el stream.

- **HTTP Server y Client**

- Beast permite construir **servidores HTTP** y **clientes HTTP** usando sockets TCP o SSL.
- Soporta **HTTP/1.1** (no HTTP/2 aún de forma nativa).

Manejo de WebSockets

- **WebSocket Stream**

- `websocket::stream<T>`: Abstrae una conexión WebSocket sobre un stream TCP o SSL.

- **Operaciones WebSocket**

- `websocket::handshake / async_handshake`: Realiza el handshake inicial para establecer la conexión.
- `websocket::read / async_read`: Recibe mensajes WebSocket.
- `websocket::write / async_write`: Envía mensajes WebSocket.
- `websocket::close`: Cierra la conexión de forma ordenada.

Manejo de WebSockets 2

- **Soporte de Frames**

- Permite enviar y recibir **frames de texto o binarios**, con control sobre fragmentación y flags.

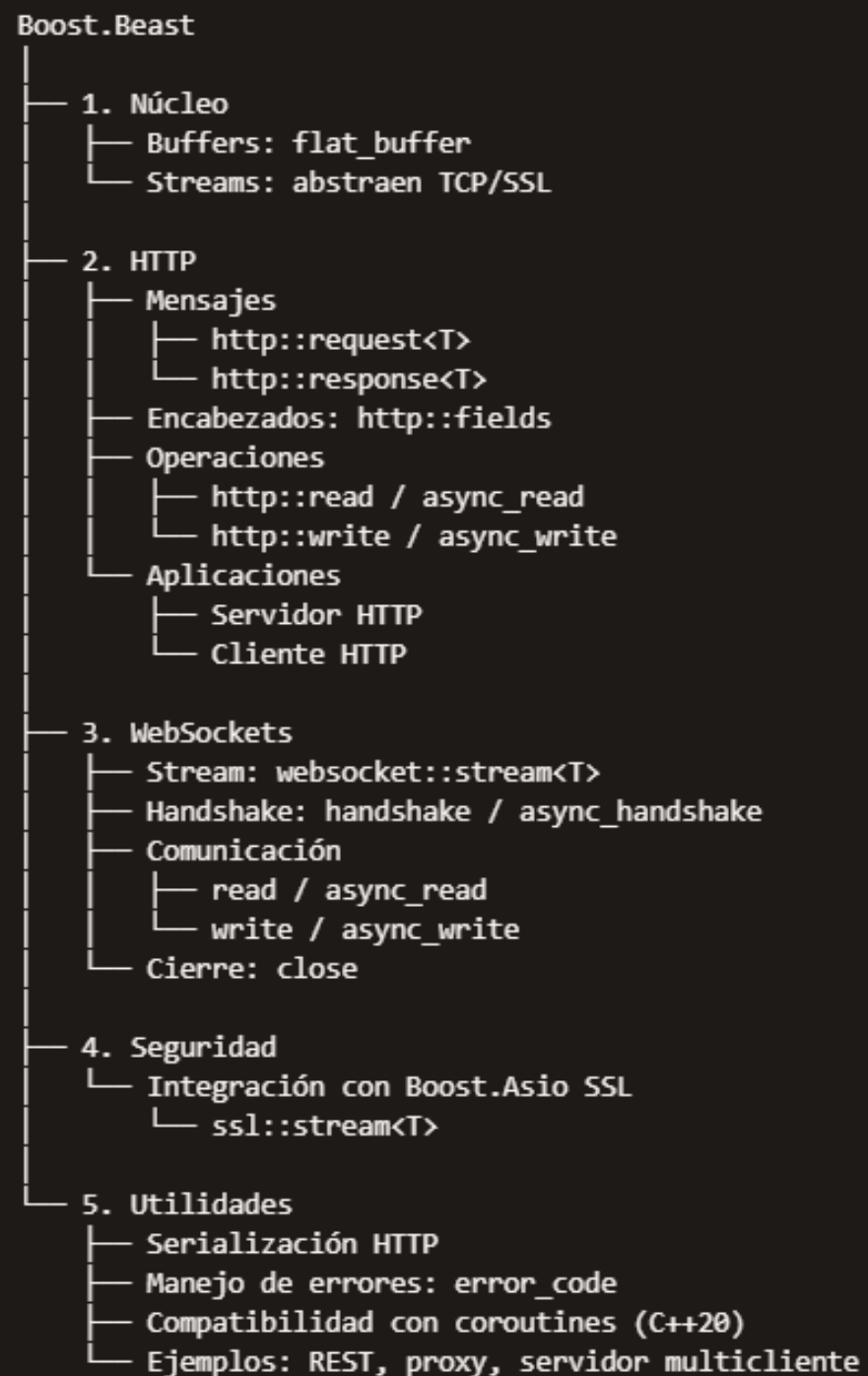
Seguridad y SSL

- Beast no gestiona SSL directamente, pero se integra perfectamente con Boost.Asio SSL.
- Puedes envolver los streams con `boost::asio::ssl::stream` para manejar conexiones seguras.

Utilidades y Extras

- Serialización y parsing de mensajes HTTP.
- Control de errores mediante `boost::system::error_code`.
- Compatibilidad con coroutines (`co_await`, `co_spawn`) en C++20.
- Ejemplos y patrones para servidores multiciente, proxies, y clientes REST.

Organización interna



Tipos de aplicaciones Http con Boost.Beast

- Se pueden implementar una gran variedad de aplicaciones de red. Manejando protocolos como Http y WebSockets.
- **1 - Cliente HTTP**
 - Realiza peticiones a servidores externos (GET, POST, PUT, DELETE...).
 - Ideal para consumir APIs REST desde C++.
 - Puedes manejar encabezados, cuerpos JSON, autenticación, etc.
 - *Ejemplo: Un cliente que consulta la API de OpenWeather y muestra el clima en Madrid.*

Tipos de aplicaciones Http con Boost.Beast

- **2 - Proxy HTTP**

- Recibe peticiones de clientes y las redirige a otros servidores.
- Puede modificar encabezados, filtrar contenido o registrar tráfico.
- *Ejemplo: Un proxy que añade autenticación a peticiones antes de reenviarlas a un backend.*

- **3 - API RESTful**

- Servidor que expone endpoints como /usuarios, /productos, etc.
- Maneja rutas, métodos HTTP y respuestas en JSON.
- Se puede integrar con bases de datos y lógica de negocio.
- *Ejemplo: Una API para gestionar inventario desde una app móvil.*

Tipos de aplicaciones Http con Boost.Beast

- **4 - Microservicio HTTP**

- Aplicación ligera que realiza una tarea específica (ej. validación, cálculo, logging).
- Se comunica con otros servicios vía HTTP o WebSocket.
- Ideal para arquitecturas distribuidas.
- *Ejemplo: Un microservicio que calcula precios con IVA y responde en milisegundos.*

- **5 - Servidor de archivos estáticos**

- Sirve HTML, CSS, JS, imágenes y otros recursos desde disco.
- Útil para alojar páginas web o documentación técnica.
- *Ejemplo: Un servidor que entrega una SPA (Single Page Application) compilada en React.*

Tipos de aplicaciones Http con Boost.Beast

- **6 - Servidor de streaming HTTP**
 - Envía datos en tiempo real usando chunked encoding o SSE (Server-Sent Events).
 - Útil para dashboards, logs en vivo o feeds de eventos.
 - *Ejemplo: Un servidor que transmite métricas de sensores cada segundo.*
- **7 - Servidor de autenticación**
 - Maneja login, tokens JWT, sesiones y autorización.
 - Puede integrarse con OAuth2, LDAP o bases de datos.
 - *Ejemplo: Un backend que valida credenciales y emite tokens para apps cliente.*
- ***WebSocket estaría más enfocado para la comunicación en tiempo real.***

Peticiones Http

Ejemplo: Servidor Http

- Implementar un servidor Http:
 - Tipos utilizados
 - Pasos para implementar el Servidor

Servidor Http - Tipos

- **boost::asio::io_context**
 - Es el motor de eventos de Boost.Asio.
 - Gestiona operaciones de entrada/salida (I/O), como aceptar conexiones o leer datos.
 - Se usa para crear el acceptor y los sockets.

Servidor Http - Tipos

- **boost::asio::ip::tcp::acceptor**

- Se encarga de escuchar en un puerto TCP.
- Espera conexiones entrantes y las acepta, creando un socket para cada cliente.

- **boost::asio::ip::tcp::socket**

- Representa una conexión TCP con un cliente.
- Se usa para leer la petición HTTP y enviar la respuesta.

Servidor Http - Tipos

- **boost::beast::flat_buffer**
 - Buffer de Beast para almacenar datos leídos del socket.
 - Necesario para las operaciones de lectura HTTP.
- **boost::beast::http::request<T> y http::response<T>**
 - Representan mensajes HTTP.
 - El tipo T indica el tipo de cuerpo (para un mensaje, string_body para texto plano).
 - Se usan para leer la petición del cliente y construir la respuesta.

Servidor Http - Tipos

- **boost::beast::http::read y http::write**
 - Funciones para leer una petición HTTP desde el socket y escribir una respuesta.
 - Son operaciones sincrónicas, pero también existen versiones asincrónicas (async_read, async_write).

Pasos para implementar el Servidor Http I

- **1 - Inicialización**

- `HttpServer server(ioc, 8080);`

- **2 - Escucha de conexiones:**

- `tcp::acceptor acceptor(ioc, tcp::endpoint(tcp::v4(), port));`
- El acceptor se configura para escuchar en IPv4 y en el puerto indicado

- **3 - Bucle de atención:**

- `for (;;) {`
- `tcp::socket socket(ioc_);`
- `acceptor_.accept(socket);`
- `handle_request(socket);`
- `}`

El servidor entra en un bucle infinito:

Acepta una conexión entrante.

Crea un socket para comunicarse con el cliente.

Llama a `handle_request()` para procesar la petición.

Pasos para implementar el Servidor Http II

- **4 – Procesamiento de la petición:**

- `http::request<http::string_body> req;`
- `http::read(socket, buffer, req);`
- Se lee la petición HTTP del cliente.
- Se almacena en el objeto req.

Pasos para implementar el Servidor Http III

- 5 – Construcción de la respuesta:
 - `http::response<http::string_body> res{http::status::ok, req.version()};`
 - `res.set(http::field::server, "Beast/1.0");`
 - `res.set(http::field::content_type, "text/plain");`
 - `res.body() = "Mensaje desde el Servidor";`
 - `res.prepare_payload();`
- *Se crea una respuesta con estado 200 OK.*
- *Se añaden encabezados como Server y Content-Type.*
- *Se asigna el cuerpo del mensaje.*
- *prepare_payload() calcula automáticamente el tamaño del cuerpo.*

Pasos para implementar el Servidor Http IV

- `http::write(socket, res);`
- Se envía la respuesta al cliente a través del socket.

WebSockets

WebSockets

- Dentro de la librería boost.beast con **WebSockets** podemos implementar:
 - Un **cliente WebSocket**: conectar con un servidor WebSocket remoto (para consumir datos en tiempo real).
 - Un **Servidor WebSocket**: crear un servidor que escuche conexiones WebSocket entrantes y gestionar múltiples clientes simultáneamente.

Realizar el handshake

- Gestiona el **handshake HTTP** inicial que convierte una conexión HTTP en una conexión WebSocket.
- Permite personalizar los encabezados del handshake para añadir autenticación, tokens, etc.

Enviar y recibir mensajes

- Lectura y escritura síncrona (read, write)
- Lectura y escritura asíncrona (async_read, async_write)
- Compatible con mensajes de texto y binarios
- Se pueden utilizar buffers dinámicos o estáticos, y gestionar los mensajes con precisión.

Soporte para WebSocket seguro (WSS)

- Integración con SSL / TLS mediante Boost.Asio
- Establecer conexiones seguras usando certificados y clases privadas
- Ideal para aplicaciones que requieren confidencialidad (como chats, trading, IoT)

Control de flujo y gestión de errores

- Manejo de errores detallado con `boost::system::error_code`
- Control de cierre de conexión (close) con códigos estándar WebSocket
- Detección de desconexiones, timeouts, y errores de protocolo

Personalización avanzada

- Puedes acceder directamente a los **encabezados HTTP** del handshake
- Configurar opciones como:
 - **Fragmentación de mensajes**
 - **Tamaño máximo de buffer**
 - **Control de ping/pong** para mantener viva la conexión

Integración con otras tecnologías

- Compatible con **Boost.Asio coroutines** (co_spawn, awaitable)
- Puedes combinar WebSockets con **HTTP/REST**, **TCP**, o **SSL** en una misma aplicación
- Ideal para servidores híbridos que ofrecen tanto APIs como canales WebSocket

Ejemplos de uso

- Chat en tiempo real
- Streaming de datos financieros
- Juegos multijugador
- Comunicación entre dispositivos IoT
- Actualización en vivo de interfaces web.

Ejemplo: Cliente WebSocket con Boost.Beast

- Pasos:
 - Resolver el host y el puerto
 - Establecer la conexión TCP
 - Realiza el handshake WebSocket
 - Enviar un mensaje
 - Recibe la respuesta
 - Cierra la conexión
- Disponemos de un servidor público por el puerto 80:
- **echo.websocket.events**
- Para probar los WebSockets

- `asio::io_context ioc; // Definir el contexto de in-out`
- `tcp::resolver resolver(ioc); // Resolver DNS:`
- **`auto const results = resolver.resolve("echo.websocket.events", "80");`**
- `websocket::stream<tcp::socket> ws(ioc); // Crear el WebSocket:`
- **`// Conectar al Servidor: intenta conectar con el primer endpoint disponible`**
- `asio::connect(ws.next_layer(), results.begin(), results.end()); ws.handshake("echo.websocket.events", "/"); // Handshake WebSocket:`
- `std::string msg = "Mensaje de Boost.Beast"; // Enviar el mensaje:`
- `ws.write(asio::buffer(msg));`
- **`beast::flat_buffer buffer; // Leer la respuesta:`**
- `ws.read(buffer);`
- `std::cout << "Respuesta del Servidor: " << beast::make_printable(buffer.data()) << std::endl;`
- **`beast::flat_buffer buffer2; // Leer una segunda respuesta:`**
- `ws.read(buffer2);`
- `std::cout << "Respuesta 2 del Servidor: " << beast::make_printable(buffer2.data()) << std::endl;`
- `ws.close(websocket::close_code::normal); // Cerrar la conexión:`

results

- **results** es un objeto de tipo **tcp::resolver::results_type**, que es básicamente una colección de **tcp::endpoint + metadatos**.
- De cada elemento representa una posible dirección IP y puerto a la que puedes conectarte.
- De cada elemento puedes extraer:
- Dirección IP (`endpoint.address()`): Por ejemplo, 93.184.216.34
- Puerto (`endpoint.port()`): En este caso, 80 (puerto HTTP)
- Familia de protocolo (`endpoint.protocol()`): Por ejemplo, `tcp::v4()` o `tcp::v6()`
- Nombre del host y servicio (si accedes a los metadatos): Puedes obtener el nombre original que se resolvió (`host_name()`, `service_name()`)

Inspeccionar

```
for (auto const& entry : results) {  
    auto endpoint = entry.endpoint();  
    std::cout << "IP: " << endpoint.address().to_string()  
        << ", Puerto: " << endpoint.port()  
        << ", Protocolo: " << (endpoint.protocol() == tcp::v4() ? "IPv4" : "IPv6")  
        << std::endl;  
}
```

Ejemplo: Servidor WebSocket con Boost.Beast

- Este servidor:
 - Escucha en un puerto TCP.
 - Acepta conexiones WebSocket.
 - Lee mensajes del cliente.
 - Los devuelve tal cual (eco).
 - Cierra la conexión cuando el cliente lo solicita.

Testear Servidor WebSocket

- Disponemos de una herramienta online para testear el servidor:
- <https://piehost.com/websocket-tester>
- Y luego nos conectamos a:
- ws://localhost:8080

Web Socket Secure

WSS

WSS (WebSocket Secure)

- Necesitamos hacer algunos cambios:
 - Crear un contexto `ssl:context`
 - Cambiar el tipo WebSocket stream
 - Realizar el handshake SSL antes del handshake WebSocket
 - El puerto tiene que ser 443
-
- Necesitamos la herramienta **openssl**, y generar un certificado

Tipos de certificado

Tipo de certificado	¿Quién lo emite?	¿Dónde se usa?	¿Requiere hardware?
DNI electrónico (DNLe)	Gobierno (Policía Nacional en España)	Trámites oficiales, firma electrónica	Sí, lector de tarjetas
Certificado FNMT	Fábrica Nacional de Moneda y Timbre	Administración pública, firma digital	No
Certificado local (OpenSSL)	Tú mismo con OpenSSL	Desarrollo, pruebas, servidores propios	No
Certificados SSL/TLS	Autoridades de certificación (CA)	Sitios web seguros (HTTPS)	No
Certificados de firma de código	CA como DigiCert, Sectigo	Firmar software, garantizar integridad	No

CA: Autoridad de certificación

openssl

- Comando: ***openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365***
- El comando:
 - Crea un certificado autofirmado (no emitido por una CA).
 - Clave privada RSA de 2048 bits
 - Pide una serie de datos
- Así como una contraseña para encriptar la clave:
 - Se genera como resultado dos archivos:
 - **key.pem → clave privada**
 - **cert.pem → tu certificado público autofirmado**
 - **Estos dos ficheros son necesarios para WSS**

openssl

- Permite activar HTTPS / WSS
- Cifrar comunicaciones entre dispositivos
- Probar servicios sin tener que tener certificados oficiales
- Al comando se le puede añadir un parámetro para evitar que nos pida la información
- -subj “...”

openssl

Campo	Descripción
C	País (Country)
ST	Estado o provincia (State)
L	Localidad o ciudad (Locality)
O	Organización (Organization)
OU	Unidad organizativa (Organizational Unit)
CN	Nombre común (Common Name)
emailAddress	Correo electrónico

- Cada inicial lleva una barra / delante del campo.
- En el comando si añadimos **-nodes** no encripta la clave privada **key.pem**

```
openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365 -nodes \  
-subj "/C=ES/ST=Madrid/L=Madrid/O=AntonioTech/OU=IoT/CN=raspberry.local/emailAddress=antonio@ex.com"
```

Servidor WSS

- **El primer paso es configurar el contexto para SSL.**
- Tenemos que indicar los dos ficheros generados anteriormente.

```
boost::asio::ssl::context ctx(boost::asio::ssl::context::tlsv12);
```

```
ctx.set_options(  
    boost::asio::ssl::context::default_workarounds |  
    boost::asio::ssl::context::no_sslv2 |  
    boost::asio::ssl::context::no_sslv3 |  
    boost::asio::ssl::context::single_dh_use  
);
```

```
ctx.use_certificate_file("cert.pem", boost::asio::ssl::context::pem);  
ctx.use_private_key_file("key.pem", boost::asio::ssl::context::pem);
```

Diferencias entre: `boost::asio::io_context` / `boost::asio::ssl::context`

- **`boost::asio::io_context`**

- El motor principal de I/O (asíncronas)
- Coordinar eventos en conexiones TCP, temporizadores y lectura / escritura de sockets.
- Ejecutar los handlers cuando ocurren eventos

- **`boost::asio::ssl::context`**

- Contexto de configuración para TLS/SSL, para cifrar comunicaciones con HTTPS / WSS
- Certificados a utilizar
- Que claves privadas cargar
- Protocolos TLS como TLS 1.2 o 1.3)
- Opciones de seguridad

```
void run_server_1_mensaje(net::io_context& ioc, ssl::context& ctx, unsigned short port) {  
    tcp::acceptor acceptor(ioc, tcp::endpoint(tcp::v4(), port));
```

Servidor WSS

```
for (;;) {  
    tcp::socket socket(ioc);  
    acceptor.accept(socket);  
  
    ssl::stream<tcp::socket> ssl_stream(std::move(socket), ctx);  
    ssl_stream.handshake(ssl::stream_base::server);  
  
    websocket::stream<ssl::stream<tcp::socket>> ws(std::move(ssl_stream));  
    ws.accept();  
  
    beast::flat_buffer buffer;  
    ws.read(buffer);  
    ws.text(ws.got_text());  
    std::cout << "Mensaje recibido: " << beast::make_printable(buffer.data()) << std::endl;  
  
    ws.write(buffer.data());  
}
```

Servidor WSS - Pasos

- **Pasos**

- **1 – Inicializar io_context, configurar SSL/TLS e indicar el puerto**

- **2 – Inicializar el servidor**

- `tcp::acceptor acceptor(ioc, tcp::endpoint(tcp::v4(), port));`
- Crea un acceptor TCP para escuchar por el puerto indicado
- Y utilizamos IPv4

- **3 – Bucle principal para aceptar clientes:**

- `for (;;) {`
- `tcp::socket socket(ioc); // Acepta conexiones`
- `acceptor.accept(socket); // y crea un socket TCP`

Servidor WSS – Pasos II

- **4 – HandShake TLS**

- **// Crea un stream SSL sobre TCP**
- `ssl::stream<tcp::socket> ssl_stream(std::move(socket), ctx);`
- **// Realiza el handshake TLS como servidor**
- `ssl_stream.handshake(ssl::stream_base::server);`

- **5 – HandShake WebSocket**

- **// Crea un stream WebSocket sobre el canal TLS**
- `websocket::stream<ssl::stream<tcp::socket>> ws(std::move(ssl_stream));`
- **// Realiza el handshake WebSocket, para completar la conexión WSS**
- `ws.accept();`

Servidor WSS – Pasos III

- **6 – lectura del mensaje**

- **// El buffer se utiliza para los datos entrantes**
- `beast::flat_buffer buffer;`
- **// Lee un mensaje del buffer.**
- `ws.read(buffer);`

- **7 – Procesamiento y eco**

- `// Configura el mensaje para tratarlo como texto no binario`
- `ws.text(ws.got_text());`
- **// Imprime el mensaje recibido en un formato legible**
- `std::cout << "Mensaje recibido: " << beast::make_printable(buffer.data()) << std::endl;`
- **// De Vuelta al cliente, hace el eco**
- `ws.write(buffer.data());`

Cliente WSS - Pasos

- Utilizaríamos: **Boost.Beast + Boost.Asio + OpenSSL**
- **Pasos:**
 - **1- Configurar el contexto SSL**
 - `boost::asio::ssl::context ctx(boost::asio::ssl::context::tlsv12);`
 - **// Para certificados autofirmados**
 - `ctx.set_verify_mode(boost::asio::ssl::verify_none);`
 - **2- Resolver y conectar**
 - `boost::asio::io_context ioc;`
 - `tcp::resolver resolver(ioc);`
 - `auto const results = resolver.resolve("localhost", "9002");`
 - `boost::asio::ssl::stream<tcp::socket> stream(ioc, ctx);`
 - `boost::asio::connect(stream.next_layer(), results);`
 - `stream.handshake(boost::asio::ssl::stream_base::client);`

Cliente WSS – Pasos II

- **Pasos:**

- **3 – Handshake WebSocket**

- `beast::websocket::stream<boost::asio::ssl::stream<tcp::socket>>`
`ws(std::move(stream));`
 - `ws.handshake("localhost", "/");`

- **4 – Enviar y recibir**

- `ws.write(boost::asio::buffer("Hola servidor"));`
 - `beast::flat_buffer buffer;`
 - `ws.read(buffer);`
 - `std::cout << "Respuesta: " << beast::make_printable(buffer.data()) << std::endl;`

Librería crow

Contenidos

- Librería crow
 - Crear servidores HTTP
 - Diseñar APIs RESTful
 - Enviar y recibir JSON
 - Soporte para WebSockets
 - Integración con Bases de datos

Introducción

- **Crow** es un microframework web para C++ que te permite construir **APIs RESTful**, **servidores HTTP**, y hasta **WebSockets**,
- Sintaxis muy parecida a frameworks como Flask (Python) o Express (Node.js).
 - **Header-only**: no necesitas compilar la librería, solo incluirla.
 - **Rápido**: diseñado para alto rendimiento.
 - **Multihilo**: soporta múltiples hilos para manejar peticiones concurrentes.
 - **JSON integrado**: sin necesidad de librerías externas para manejar JSON.

Instalación

- `vcpkg install crow`
- `vcpkg integrate install`
- Sirve para:
 - Definir operaciones CRUD
 - Servidores HTTP
 - WebSockets
 - Microservicios
 - Integración con Base de datos

Ejemplo

```
#include "crow_all.h"
```

```
int main() {
```

```
    crow::SimpleApp app;
```

```
    CROW_ROUTE(app, "/")([](){
```

```
        return "¡Hola desde Crow!";
```

```
});
```

```
    CROW_ROUTE(app, "/json")([](){
```

```
        crow::json::wvalue x;
```

```
        x["mensaje"] = "Hola mundo";
```

```
        return x;
```

```
});
```

```
    app.port(8080).multithreaded().run();
```












```
}
```


Comparación y selección de herramientas según el tipo de aplicación

Cuando utilizar una u otra tecnología

- **WebSocket** ideal cuando necesitamos:
 - Comunicación bidireccional en tiempo real.
 - Actualizaciones instantáneas sin tener que hacer **polling** constante.
 - Menor sobrecarga que HTTP con conexiones persistentes.

boost.beast vs crow

Característica	Crow	Boost.Beast
 Nivel de abstracción	Alto (microframework estilo Flask/Express)	Bajo (control total sobre HTTP y sockets)
 Multihilo	Sí, integrado	No directamente, debes usar Boost.Asio
 JSON integrado	Sí (con su propio sistema)	No, debes usar una librería externa
 Routing	Sí, muy sencillo (<code>CROW_ROUTE</code>)	No, lo implementas tú manualmente
 RESTful API	Ideal para montar APIs rápidamente	Requiere más trabajo para estructurar rutas
 Control de bajo nivel	Limitado (Crow abstrae mucho)	Total (manejas buffers, headers, etc.)
 WebSockets	Soportado	Soportado (más flexible)
 Documentación	Clara y sencilla	Técnica y extensa
 Dependencias	Ligera (header-only)	Requiere Boost completo
 Testing y depuración	Fácil de probar con mocks	Más complejo, pero más preciso
 Aprendizaje	Rápido para principiantes	Requiere experiencia con Boost.Asio

Elegir crow cuando

- Quieres montar una API RESTful rápida y sencilla.
- Prefieres una sintaxis clara y moderna.
- No necesitas control de bajo nivel sobre sockets o buffers.
- Estás trabajando en un microservicio o backend ligero.
- Quieres algo que funcione bien en Windows y Linux con vcpkg.

Elegir Boost.Beast

- Necesitas control total sobre el protocolo HTTP.
- Estás construyendo un servidor de alto rendimiento o personalizado.
- Quieres integrar con otros componentes de Boost (Asio, SSL, etc.).
- Estás trabajando en aplicaciones que requieren WebSockets avanzados.
- Buscas máxima flexibilidad y rendimiento.

Ejemplo

- Para un CRUD de una entidad de la BD con rutas del estilo: /empleados/<id> y respuesta en JSON: **crow**
- Para un servidor que maneja miles de conexiones simultaneas, con control de headers, trailers y buffers: **Boost.beast**