

Creación de APIs RESTful con C++

Antonio Espín Herranz

Contenidos

- Implementación de APIs en C/C++:
 - Uso de frameworks como **Cpp-REST-SDK**, Crow o **Pistache**.
 - Creación de rutas para GET, POST, PUT, DELETE.
 - Manejo de respuestas HTTP y serialización/deserialización de datos en JSON.
- Gestión de errores y excepciones:
 - Manejo de códigos de error y optimización del flujo de datos.
 - Implementación de registros de errores y mensajes de diagnóstico.

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.

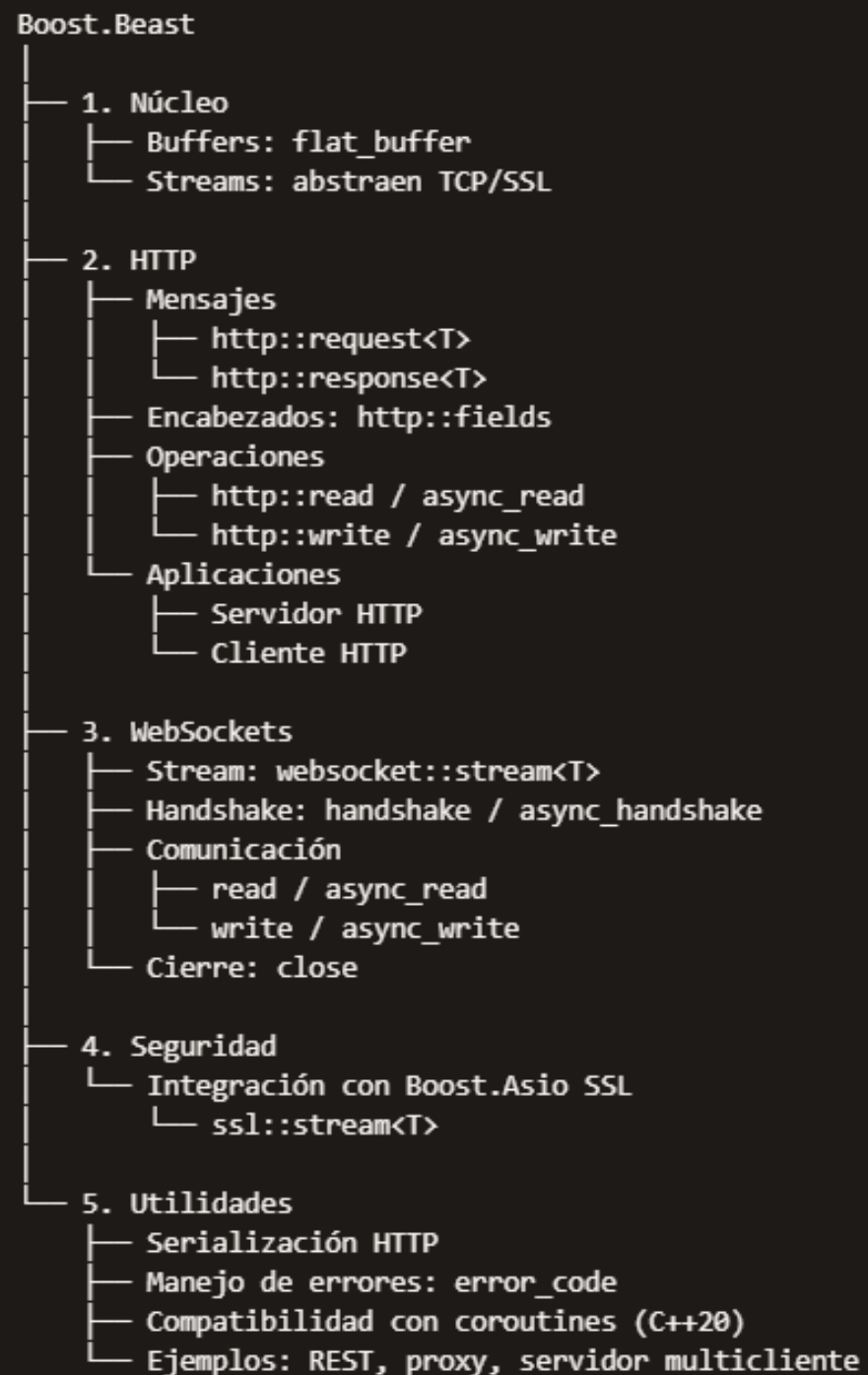
Seguridad y SSL

- 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.

Librería crow

Ojo, los proyectos con versión **C++ 17**

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 <iostream>
```

```
#include <crow.h>
```

```
void testCrow() {
```

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

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

```
        return "Hello world";
```

```
    });
```

```
    app.port(18080).run();
```

```
}
```

```
int main(){
```

```
    testCrow();
```

```
    return 0;
```

```
}
```

Respuesta en json

<http://localhost:8080/json>

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

```
CROW_ROUTE(app, "/json")([]() {  
    crow::json::wvalue respuesta;  
    respuesta["mensaje"] = "¡Hola desde Crow!";  
    respuesta["estado"] = "ok";  
    respuesta["codigo"] = 200;  
    return crow::response{ respuesta };  
});
```

```
app.bindaddr("127.0.0.1");  
app.port(8080);  
app.concurrency(2);  
app.run();
```

Operaciones CRUD

- Create → POST → `crow::HTTPMethod::POST`
 - Subir un recurso en el servidor.
 - El recurso se envía en el body
- Read → GET → `crow::HTTPMethod::GET`
 - Recuperar un recurso del servidor
 - Normalmente se enviará un parámetro: id en la URL
- Update → PUT → `crow::HTTPMethod::PUT`
 - Actualizar un recurso del servidor
 - El recurso se envía en el body
- Delete → DELETE → `crow::HTTPMethod::Delete`
 - Eliminar un recurso del servidor
 - Normalmente se enviará un parámetro: id en la URL

Parámetros en las peticiones

- Necesitamos un objeto: **crow::SimpleApp app;**
- Sobre este se van definiendo las rutas y los métodos:
- **CROW_ROUTE(app, /ruta").methods(método)(lambda) {}**

```
CROW_ROUTE(app, "/ruta")  
    .methods("GET"_method, "POST"_method) // Opcional: especifica  
    métodos  
    ([](const crow::request& req){  
        // Aquí va tu lógica  
        return crow::response("Hola mundo");  
    });
```

app: instancia de `crow::SimpleApp`

"/ruta": URL que responde

req: objeto que contiene headers, cuerpo, método, etc.

Formato de las lambdas

- Sin parámetros:

```
CROW_ROUTE(app, "/ping")  
([](){  
    return "pong";  
});
```

- Con acceso al cuerpo de la petición:

```
CROW_ROUTE(app, "/echo").methods("POST"_method)  
([](const crow::request& req){  
    return crow::response(req.body);  
});
```

Con parámetros en la URL

- CROW_ROUTE(app, "/saludo/<**string**>")
- ([](std::string nombre){
- return "Hola " + nombre;
- });

Tipo de los parámetros en las peticiones

Tipo	Ejemplo de ruta	Conversión automática
<code>int</code>	<code>/suma/<int>/<int></code>	Sí
<code>uint64_t</code>	<code>/id/<uint64_t></code>	Sí
<code>double</code>	<code>/precio/<double></code>	Sí
<code>float</code>	<code>/temperatura/<float></code>	Sí
<code>std::string</code>	<code>/saludo/<string></code>	Sí
<code>char</code>	<code>/letra/<char></code>	Sí
<code>bool</code>	<code>/activo/<bool></code>	Sí (<code>true</code> / <code>false</code>)

`/productos/<int>/<double>/<string>`

Tipos de parámetros en las peticiones

- Si el tipo no se puede convertir desde la URL, Crow devuelve automáticamente un error 404.
- Los parámetros se extraen en el orden en que aparecen en la ruta.
- No se puede usar tipos complejos directamente (como `std::vector`, `struct`, etc.) en rutas, pero sí dentro del cuerpo JSON de la petición.
- Los parámetros definidos, tendrán que ser luego parámetros en las funciones lambda.

Asociación rutas → lambdas

Tipo	Ejemplo de ruta	Conversión automática
<code>int</code>	<code>/suma/<int>/<int></code>	Sí
<code>uint64_t</code>	<code>/id/<uint64_t></code>	Sí
<code>double</code>	<code>/precio/<double></code>	Sí
<code>float</code>	<code>/temperatura/<float></code>	Sí
<code>std::string</code>	<code>/saludo/<string></code>	Sí
<code>char</code>	<code>/letra/<char></code>	Sí
<code>bool</code>	<code>/activo/<bool></code>	Sí (<code>true</code> / <code>false</code>)

Si la lambda tiene que recibir la request tiene que ser el primer parámetro

Con JSON

```
CROW_ROUTE(app, "/json").methods("POST"_method)
([])(const crow::request& req){
    auto datos = crow::json::load(req.body);
    if (!datos)
        return crow::response(400);

    std::string nombre = datos["nombre"].s(); // A string
    return crow::response("Hola " + nombre);
});
```

Con JSON

- Los datos que vienen por la request en json se cargan con:
 - `auto datos = crow::json::load(req.body);`
 - Devuelve un objeto de tipo: **`crow::json::rvalue`**
- Si los datos enviados son:

```
{  
  "nombre": "Ana",  
  "edad": 42  
}
```
- Para utilizarlos en C++:
 - `std::string nombre = datos["nombre"].s();` `// Extrae "Ana"`
 - `int edad = datos["edad"].i();` `// Extrae 42 como entero`

Otros métodos relacionados

Método	Tipo JSON esperado	C++ equivalente
<code>.s()</code>	string	<code>std::string</code>
<code>.i()</code>	integer	<code>int</code>
<code>.d()</code>	decimal/float	<code>double</code>
<code>.b()</code>	boolean	<code>bool</code>
<code>.t()</code>	timestamp	<code>time_t</code>

Tipos relacionados con JSON en crow

Tipo	¿Qué es?	¿Cuándo se usa?
<code>crow::json::rvalue</code>	Representa un valor JSON leído (resultado de <code>json::load</code>)	Para acceder a campos del JSON recibido
<code>crow::json::wvalue</code>	Representa un valor JSON que vas a construir o devolver	Para crear respuestas JSON
<code>crow::json::json_return_type</code>	Alias para <code>crow::json::wvalue</code>	Se usa como tipo de retorno en rutas que devuelven JSON

Ejemplos

- **rvalue** para **extraer** datos del JSON
 - `auto datos = crow::json::load(req.body);`
 - `std::string nombre = datos["nombre"].s();`
- **wvalue** para **construir** un JSON para devolverlo como respuesta
 - `crow::json::wvalue respuesta;`
 - `respuesta["estado"] = "ok";`
 - `respuesta["codigo"] = 200;`
 - `return crow::response(respuesta);`

Accesso a headers, query params

```
CROW_ROUTE(app, "/info")
([](const crow::request& req){
    std::string agente = req.get_header_value("User-Agent");
    std::string param = req.url_params.get("id"); // /info?id=123
    return crow::response("Agente: " + agente + ", ID: " + param);
});
```

Puesta en marcha del Servidor

```
#include <crow.h>
```

```
int main() {
```

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

```
    CROW_ROUTE(app, "/")
```

```
    ([](){
```

```
        return "¡Servidor Crow en marcha!";
```

```
});
```

```
    app.port(18080)
```

```
    // Puerto donde escucha el servidor
```

```
    .concurrency(4)
```

```
    // Número de hilos (multithreading)
```

```
    .multithreaded()
```

```
    // Activa el modo multihilo
```

```
    .run();
```

```
    // Arranca el servidor
```

```
}
```

Puesta en marcha Servidor

- **app.port(<número>)**
 - Define el **puerto TCP** donde el servidor escucha.
 - Ejemplo: app.port(8080) → accesible en <http://localhost:8080>
- **app.concurrency(<número>)**
 - Número de **hilos de ejecución** que Crow usará para manejar peticiones simultáneas.
 - **Valor recomendado: igual al número de núcleos del procesador.**
 - Ejemplo: app.concurrency(4) en una Raspberry Pi de 4 núcleos.
- **app.multithreaded()**
 - Activa el **modo multihilo**.
 - Si no lo usas, Crow funcionará en modo monohilo (una petición a la vez).
- **app.run()**
 - Lanza el servidor y lo deja escuchando indefinidamente.

Consideraciones

- Si no usamos **multithreaded()**
 - **crow** seguirá funcionando, pero solo podrá atender **una petición a la vez**, lo cual no es ideal si esperas concurrencia o múltiples clientes.

```
app.port(8080)
  .concurrency(std::thread::hardware_concurrency())
  .multithreaded()
  .loglevel(crow::LogLevel::Debug)
  .run();
```

- Se puede activar los logs con un nivel de log, pero son automáticos.

Niveles de log

Nivel	Descripción
Debug	Máxima información (ideal para desarrollo)
Info	Eventos importantes
Warning	Cosas que podrían causar problemas
Error	Fallos que afectan el funcionamiento
Critical	Errores graves que requieren atención inmediata

Mensajes de log

- **crow** usa **macros** para emitir logs personalizados:
 - CROW_LOG_DEBUG << "Mensaje de depuración";
 - CROW_LOG_INFO << "Servidor iniciado correctamente";
 - CROW_LOG_WARNING << "Parámetro inesperado recibido";
 - CROW_LOG_ERROR << "Error al procesar la petición";
 - CROW_LOG_CRITICAL << "Fallo grave en el sistema";

Protocolos HTTP, HTTPs y WebSockets

HTTP

- Es el protocolo base de la web. Permite la comunicación entre clientes (como navegadores) y servidores mediante un modelo solicitud-respuesta.
- Características:
- Unidireccional: El cliente envía una solicitud, el servidor responde.
- Sin estado: No guarda información entre solicitudes (aunque puede usarse con cookies o sesiones).
- Usa el puerto 80 por defecto.
- Formato textual: Las solicitudes y respuestas son legibles y estructuradas.

HTTPs

- Http secure
 - Es la versión segura de HTTP. Utiliza TLS/SSL para cifrar la comunicación entre cliente y servidor.
- Características:
 - Cifrado de extremo a extremo: Protege datos sensibles como contraseñas o tarjetas.
 - Autenticación: Verifica que estás hablando con el servidor correcto mediante certificados digitales.
 - Integridad: Evita que los datos sean modificados durante la transmisión.
 - Usa el puerto 443 por defecto.

WebSockets

- Es un protocolo de comunicación **bidireccional y persistente** que permite que cliente y servidor intercambien datos en tiempo real sin necesidad de múltiples solicitudes HTTP
- Características:
 - Full-duplex: Ambos lados pueden enviar y recibir datos simultáneamente.
 - Conexión persistente: Se mantiene abierta, ideal para apps en tiempo real.
 - Menor latencia: Los datos se envían tan pronto como están disponibles.
 - Usa el puerto 80 (ws://) o 443 (wss://) dependiendo de si es seguro.

WebSockets

- **Casos de uso:**
 - Chats en vivo
 - Juegos multijugador
 - Notificaciones en tiempo real
 - Streaming de datos

Comparativa

Protocolo	Dirección	Persistencia	Seguridad	Ideal para...
HTTP	Cliente → Servidor	No	No	Páginas web estáticas
HTTPS	Cliente → Servidor	No	Sí	Formularios, login, e-commerce
WebSockets	Bidireccional	Sí	Opcional	Apps en tiempo real

Peticiones HTTP

Verbo	Acción principal	Equivalente CRUD	Uso típico en APIs REST
GET	Obtener datos	Read	Consultar información (ej. productos)
POST	Crear nuevos datos	Create	Registrar usuario, añadir producto
PUT	Actualizar datos existentes	Update	Modificar perfil, cambiar contraseña
DELETE	Eliminar datos	Delete	Borrar cuenta, eliminar comentario

Ejemplos

- **GET** /libros → devuelve la lista de libros.
- **GET** /libros/123 → devuelve el libro con ID 123.
- **POST** /libros → crea un nuevo libro (con datos en el cuerpo).
- **PUT** /libros/123 → actualiza el libro con ID 123.
- **DELETE** /libros/123 → elimina el libro con ID 123.

Serialización / Deserialización con JSON

Parsear datos en Json

- Librería: **nlohmann-json**
- **vcpkg install nlohmann-json**
- **vcpkg integrate install**
- En el proyecto hacemos referencia a:
- **#include <nlohmann/json.hpp>**


```
#include <iostream>
#include <sstream>
#include <nlohmann/json.hpp>

void testJson() {
    nlohmann::json doc;

    doc["curso"] = "Microservicios en C++";
    doc["horas"] = 25;
    doc["tecnologias"] = { "xml", "json", "rest", "soap" };
    std::cout << "Json: " << doc.dump(4) << std::endl;
    std::cout << "curso: " << doc["curso"] << std::endl;
}
```

```
void strToJson() {  
    // Se define una cadena Raw: con formato json  
    std::stringstream ss(R"({"nombre":"Ana","edad":28,  
        "intereses":["programacion","musica","senderismo"]}");  
    nlohmann::json doc;  
  
    // Se convierte a json:  
    ss >> doc;  
    std::cout << "nombre: " << doc["nombre"] << std::endl;  
    std::cout << doc << std::endl;  
}
```

Vector / Grabar a fichero

- Se pueden definir vectores del tipo: **nlohmann::json** para almacenar objetos json:
 - `std::vector<nlohmann::json> array;`
 - `nlohmann::json grupo;`
 - `grupo = array;` // Se convierte automáticamente
 - **`grupo.dump(4)`** // Añade indentación, el resultado se puede grabar en un fichero o se imprime por la pantalla.
- Utilizar operador `<<`, con un objeto **`std::ofstream`** o **`std::cout`**

Objetos a Json

```
nlohmann::json Pedido::to_json() const
{
    return nlohmann::json{ {"idpedido", this->idpedido},
        {"cliente", this->cliente},
        {"empresa", this->empresa},
        {"empleado", this->empleado},
        {"importe", this->importe},
        {"pais", this->pais}};
}
```

De JSON a Objeto

Gestión de errores y excepciones

Manejo de códigos de error y optimización del flujo de datos.

Implementación de registros de errores y mensajes de diagnóstico












Apéndice

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**