

Programación IV
Profesora Cynthia Estrada
Fecha: 08/09

“Armando servidor HTTP”

Integrantes:

- “Díaz Rossini Juan José”,
 - “Gallo Genaro”
 - “Navarro Victor Leandro”
-

Buscar la definición, la función, el uso, la aplicación y características de:

- `Require(módulo node.js)`

- `CreateServer(método de HTTP)`

- `Req(Request)`

- `Res(Responde)`

- `Res.end`

- `Listen`

- `Puerto`

- `Callback`

1. ¿Qué es Express?

2. Características de Express

3. ¿Qué es CRUD?

4. ¿Las operaciones CRUD tienen sus equivalentes con los métodos HTTP?

5. ¿Qué es API? Características

6. ¿Cómo se qué método utilizar con respecto a la solicitud?

- `Require (módulo Node.js)`

`require` es una función nativa de Node.js que permite importar módulos, librerías o archivos dentro de un proyecto.

Su propósito es reutilizar código ya existente, evitando reescribir funciones o estructuras repetidas.

Se invoca con

`const nombre = require("módulo")`

Puede cargar tanto módulos propios de Node.js (como http, fs, path) como módulos externos instalados con npm o archivos creados por el programador.

Es fundamental para estructurar aplicaciones en múltiples archivos, facilitando la organización y mantenimiento del código.

Características:

- Es sincrónico: ejecuta la carga del módulo antes de continuar.
 - Permite importar módulos externos y propios.
 - Es la base del sistema de módulos de Node.js (CommonJS).
-

- `createServer (método de HTTP)`

Es un método del módulo http de Node.js que crea un servidor web básico. Permite escuchar peticiones HTTP (GET, POST, etc.) y generar respuestas.

```
http.createServer((req, res) => {  
  ...  
});
```

Crea el servidor y define cómo manejar cada petición entrante. Es el primer paso para construir aplicaciones web, APIs REST o servidores de prueba.

Características:

- Devuelve una instancia de servidor (`http.Server`).
 - Se basa en eventos (ejemplo: `request`, `connection`).
 - Es ligero y altamente configurable.
-

- `req (Request)`

"req" es el objeto que representa la solicitud del cliente al servidor. Contiene toda la información de la petición que el cliente hizo, como la URL, el método HTTP, los encabezados y los datos enviados. Se utiliza dentro de `createServer` para acceder a datos de la solicitud, ejemplo: `req.url`, `req.method`. Es esencial para identificar qué recurso pide el usuario y con qué método (ejemplo: acceder a `/home` con GET).

Características:

- Es un objeto de tipo `IncomingMessage`.
 - Contiene propiedades como `url`, `headers`, `method`.
 - Permite manejar formularios, parámetros o archivos enviados por el cliente.
-

- `res (Response)`

"res" es el objeto que representa la respuesta que el servidor envía al cliente. Permite configurar el mensaje que recibirá el cliente, como código de estado, encabezados y cuerpo de la respuesta. Se manipula dentro de `createServer`, por ejemplo:

```
res.writeHead(200, {  
  "Content-Type": "text/html"  
});  
res.end("Hola Mundo");
```

Se utiliza para devolver HTML, JSON, archivos u otros datos.

Características:

- Es un objeto de tipo `ServerResponse`.
 - Permite escribir datos con `res.write()` y terminar la respuesta con `res.end()`.
 - Soporta diferentes formatos (texto, JSON, binario).
-

- `res.end`

Es un método del objeto "res" que finaliza la respuesta del servidor. Indica al servidor que ya no se enviarán más datos y se cierra la comunicación con el cliente. Se emplea siempre al final de una respuesta: `res.end("Contenido enviado")`. Evita que la conexión quede abierta innecesariamente y asegura que el cliente reciba la respuesta completa.

Características:

- Puede enviar un último fragmento de datos (opcional).
 - Cierra el flujo de escritura de la respuesta.
 - Es obligatorio en cada petición para evitar bloqueos.
-

- `listen`

Es un método del servidor creado con createServer que lo pone a la escucha de conexiones entrantes. Asocia el servidor a un puerto y una dirección IP.

```
server.listen(3000, "127.0.0.1", () => {
  console.log("Servidor activo");
});
```

Es lo que hace que el servidor esté disponible para recibir peticiones.

Características:

- Recibe parámetros: puerto, dirección IP y un callback opcional.
 - Permite manejar múltiples clientes concurrentes.
 - Es no bloqueante (asincrónico).
-

- Puerto

Es un número lógico que identifica un proceso o servicio en un dispositivo dentro de una red. Permite que varias aplicaciones se comuniquen a través de una misma dirección IP diferenciando los servicios. En Node.js se suele usar **3000, 4000 o 8080** para servidores web de desarrollo.

Puerto 80: HTTP por defecto.

Puerto 443: HTTPS.

Puertos personalizados: APIs, aplicaciones internas.

Características:

- Van del rango 0 al 65535.
 - Los puertos 0-1023 son conocidos como puertos bien conocidos (requieren permisos administrativos).
 - Son esenciales para la comunicación cliente-servidor.
-

- Callback

Un **callback** es una función que se pasa como argumento a otra función y que se ejecuta una vez que esa función ha terminado su tarea. Es una de las formas más comunes de manejar la asincronía en Node.js. Permite ejecutar código después de que una operación haya finalizado, sin bloquear el flujo principal del programa. Gracias a los callbacks, Node.js puede manejar múltiples solicitudes y procesos al mismo tiempo de manera eficiente. Se usa normalmente en funciones que requieren tiempo para completarse, como la lectura de archivos, consultas a bases de datos o la creación de un servidor. Ejemplo:

```
server.listen(3000, () => {
  console.log("Servidor corriendo en el puerto 3000");
});
```

En este caso, el segundo parámetro es un callback que se ejecuta cuando el servidor empieza a escuchar en el puerto. Los callbacks son ampliamente utilizados en Node.js para:

- Definir acciones a ejecutar cuando un servidor arranca.
- Manejar respuestas de peticiones HTTP.
- Procesar resultados de operaciones de lectura/escritura de archivos.

Características:

- Pueden ser funciones anónimas o definidas previamente.
 - Son esenciales en el modelo de programación asincrónica de Node.js.
 - Pueden generar el problema conocido como **Callback Hell** si se anidan en exceso.
 - Son el antecesor de otras soluciones más modernas como **Promises y async/await**.
-

1. ¿Qué es Express?

Express es un framework minimalista para Node.js que facilita la creación de servidores web y APIs. Simplifica el manejo de rutas, peticiones y respuestas, evitando tener que programar todo manualmente con `http.createServer`. Se instala con `npm install express` y se importa con `const express = require("express")`. Es ideal para desarrollar aplicaciones web, APIs REST y microservicios.

Características:

- Permite definir rutas fácilmente (`app.get`, `app.post`, etc.).
 - Maneja middleware para procesar datos entre la petición y la respuesta.
 - Compatible con JSON y múltiples motores de plantillas.
 - Es modular y escalable.
-

2. Características de Express

- **Simplicidad:** reduce código repetitivo en comparación con `http`.
 - **Middleware:** permite agregar funciones intermedias (autenticación, validación, logs).
 - **Enrutamiento avanzado:** soporta rutas dinámicas y agrupadas.
 - **Compatibilidad:** se integra fácilmente con bases de datos y librerías externas.
 - **Ecosistema:** cuenta con gran cantidad de módulos de terceros.
-

3. ¿Qué es CRUD?

CRUD es el acrónimo de **Create**, **Read**, **Update** y **Delete**, que son las operaciones básicas para manipular datos en una base de datos o API. Permite definir la lógica principal de un sistema de gestión de información. Se aplica en prácticamente todas las aplicaciones que necesitan manejar registros, como usuarios, productos o pedidos.

- **Create:** crear un nuevo registro.
- **Read:** leer o consultar registros.
- **Update:** modificar un registro existente.
- **Delete:** eliminar un registro.

Características:

- Es la base del diseño de APIs REST.
 - Está directamente relacionado con los métodos HTTP.
-

4. ¿Las operaciones CRUD tienen sus equivalentes con los métodos HTTP?

Sí, existe una correspondencia directa entre las operaciones CRUD y los métodos HTTP:

- **Create** → POST (crear un recurso en el servidor).
- **Read** → GET (obtener datos o un recurso).
- **Update** → PUT o PATCH (modificar un recurso existente).
- **Delete** → DELETE (eliminar un recurso).

Esto permite que las APIs sean más intuitivas y fáciles de usar, ya que el método HTTP refleja claramente la acción que se está realizando.

5. ¿Qué es una API? y sus características

Una API (Application Programming Interface) es un conjunto de reglas que permite la comunicación entre dos aplicaciones o sistemas. En el caso de Node.js, se usan APIs web para que un cliente (navegador, app móvil) se comunique con un servidor.

Características:

- Define cómo interactuar con los datos y servicios.
 - Facilita la integración entre distintas plataformas.
 - Puede ser pública, privada o híbrida.
 - Las APIs modernas suelen usar JSON para transmitir información.
 - Se basan en métodos HTTP y suelen seguir la arquitectura REST.
-

6. ¿Cómo sé qué método utilizar con respecto a la solicitud?

Depende de la intención del cliente:

- ★ Si el cliente quiere consultar información, se usa GET.
- ★ Si quiere enviar nuevos datos al servidor, se usa POST.
- ★ Si busca modificar datos existentes, se usa PUT (reemplazo total) o PATCH (modificación parcial).
- ★ Si la acción es eliminar datos, se usa DELETE.

En resumen, la elección del método depende de la operación CRUD que se quiera realizar.

```
1 // 1. Importamos el módulo http con require
2 const http = require("http");
3
4 // 2. Definimos el puerto
5 const PORT = 3000;
6
7 // 3. Creamos el servidor con createServer
8 const server = http.createServer((req, res) => {
9   // 4. Configuramos la cabecera de la respuesta
10   res.writeHead(200, { "Content-Type": "text/plain" });
11
12   // 5. Dependiendo de la URL y método, respondemos distinto
13   if (req.url === "/" && req.method === "GET") {
14     res.end("Bienvenido al servidor HTTP con Node.js");
15   } else if (req.url === "/about" && req.method === "GET") {
16     res.end("Sección Acerca de...");|
17   } else if (req.url === "/data" && req.method === "POST") {
18     res.end("Datos recibidos correctamente con POST");
19   } else {
20     res.statusCode = 404;
21     res.end("Página no encontrada");
22   }
23 });
24
25 // 6. Ponemos al servidor a escuchar con listen y un callback
26 server.listen(PORT, () => {
27   console.log(`Servidor corriendo en http://localhost:${PORT}`);
28 });
29
```