



Instituto Politécnico Nacional

IPN

Escuela Superior de Cómputo

ESCOM

Academia de Redes Aplicaciones para comunicaciones  
en red

6CV2

Práctica 14

“Protocolo HTTP”

Alumna:

Navarrete Becerril Sharon Anette

Fecha de entrega: “ 25 - Noviembre - 2024”

Profesor: Ojeda Santillan Rodrigo

## OBJETIVO

Realizar un servidor HTTP que busque realizar las operaciones de tipo GET y POST.

## INTRODUCCIÓN

El protocolo HTTP (Hypertext Transfer Protocol) es el sistema esencial de comunicación para la transferencia de datos en la web. Desde su creación en 1989 por Tim Berners-Lee, HTTP se ha convertido en el pilar de la World Wide Web, permitiendo que los navegadores y servidores web se comuniquen para intercambiar recursos, como documentos HTML, imágenes, y videos. Este protocolo especifica cómo deben formatearse y transmitirse los mensajes y cómo deben actuar los servidores y clientes en respuesta a estos mensajes.

HTTP funciona bajo un modelo de cliente-servidor en el cual el cliente, generalmente un navegador, envía solicitudes y el servidor responde con los datos solicitados. Este modelo sigue una estructura de solicitud-respuesta, característica que facilita el acceso a los recursos en la web. Además, HTTP es un protocolo sin estado (stateless), lo que significa que cada solicitud es independiente, y no se almacena información de las conexiones previas. Sin embargo, permite el uso de mecanismos como la caché para mejorar la eficiencia y reducir la carga en el servidor.

Una de las fortalezas de HTTP es su simplicidad y flexibilidad. Funciona sobre TCP/IP, proporcionando una comunicación confiable, y permite la extensión mediante encabezados y métodos adicionales. A través de distintos métodos HTTP, como GET, POST y PUT, los usuarios pueden interactuar con recursos en el servidor de maneras específicas. Con el soporte de HTTPS, HTTP también ofrece opciones de autenticación y seguridad, protegiendo la información transmitida mediante cifrado SSL/TLS.

HTTP es así no solo el medio para navegar en la web, sino también una herramienta poderosa para la transferencia de datos en una amplia variedad de aplicaciones, garantizando la interoperabilidad y eficiencia de la comunicación en internet.

## DESARROLLO

Para implementar un servidor HTTP que maneje las operaciones GET y POST, se requiere tener Node.js instalado en el sistema. El primer paso es crear un archivo llamado `servidor\_http.js` y utilizar las bibliotecas incluidas por defecto con Node.js.

Este servidor HTTP se configurará para responder a solicitudes tanto GET como POST, adaptando su respuesta en función de la ruta y el método que el cliente utilice. Para lograr esto, se importan dos módulos clave de Node.js: `http`, que permite crear servidores web y gestionar solicitudes, y `url`, que facilita el análisis y procesamiento de las URLs para extraer la ruta y los parámetros de la solicitud.

El servidor se construye usando la función `http.createServer()`, la cual recibe un callback que se ejecuta con cada solicitud. Este callback toma dos parámetros: `req` (que representa la solicitud) y `res` (que representa la respuesta). En este punto, se define cómo el servidor manejará las solicitudes según la ruta y el tipo de operación solicitada.

Dentro del callback, el método `url.parse()` se utiliza para analizar `req.url` y obtener un objeto que contiene tanto la ruta (`pathname`) como los parámetros de consulta (`query`). Estos valores son fundamentales para decidir cómo responderá el servidor, según la ruta y parámetros proporcionados por el cliente.

Para identificar el método HTTP, se usa `req.method.toUpperCase()`, convirtiéndolo a mayúsculas para garantizar compatibilidad. Luego, el servidor evalúa el método y la ruta de la solicitud para decidir la respuesta adecuada.

Si el método es GET y la ruta es `/get`, el servidor toma los parámetros de la URL y los envía en la respuesta, devolviendo un código de estado 200 (indica éxito) y un cuerpo en formato JSON con un mensaje y los parámetros recibidos.

Si el método es POST y la ruta es `/post`, el servidor recoge el cuerpo de la solicitud. Dado que las solicitudes POST incluyen datos en el cuerpo, se emplea el evento `req.on('data')` para acumular la información entrante. Cuando todos los datos han sido recibidos, se activa el evento `req.on('end')`, indicando que la recolección está completa. En este caso, el servidor responde con un código de estado 200 y un cuerpo en JSON que contiene un mensaje y los datos recibidos, procesados como un objeto JavaScript.

Si la solicitud no coincide con las rutas `/get` o `/post`, el servidor responde con un código de estado 404, indicando que el recurso solicitado no se encontró. La respuesta también incluye un mensaje en JSON informando al cliente que la ruta solicitada no existe.

Una vez comprendido el funcionamiento del código se va a establecer la siguiente lógica para lograr su funcionamiento en el archivo servidor\_http.js:

```
const http = require('http');
const url = require('url');

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url, true);
  const path = parsedUrl.pathname;
  const method = req.method.toUpperCase();

  res.setHeader('Content-Type', 'application/json');

  if (method === 'GET' && path === '/get') {
    const queryParams = parsedUrl.query;
    res.statusCode = 200;
    res.end(JSON.stringify({
      message: 'Petición GET recibida',
      params: queryParams,
    }));
  } else if (method === 'POST' && path === '/post') {
    let body = '';
  }
})
```

```

req.on('data', chunk => {
    body += chunk.toString();
});

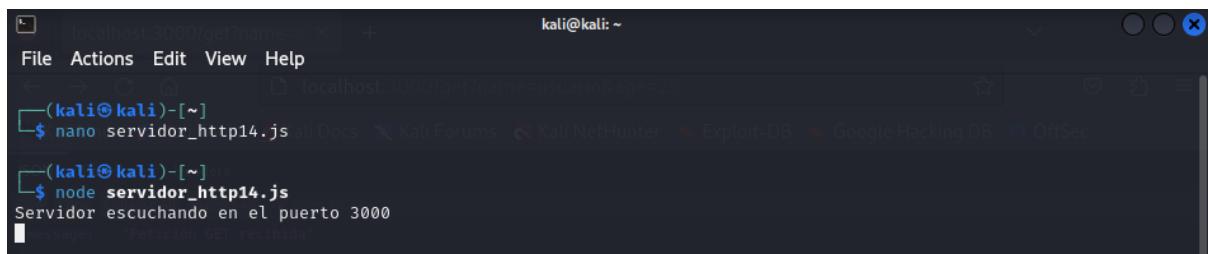
req.on('end', () => {
    res.statusCode = 200;
    res.end(JSON.stringify({
        message: 'Petición POST recibida',
        data: JSON.parse(body),
    }));
});

} else {
    res.statusCode = 404;
    res.end(JSON.stringify({
        message: 'Ruta no encontrada',
    }));
}

const PORT = 3000;
server.listen(PORT, () => {
    console.log(`Servidor escuchando en el puerto ${PORT}`);
});
}

```

Para realizar las pruebas se requiere ejecutar el servidor con el comando: `node servidor_http.js`



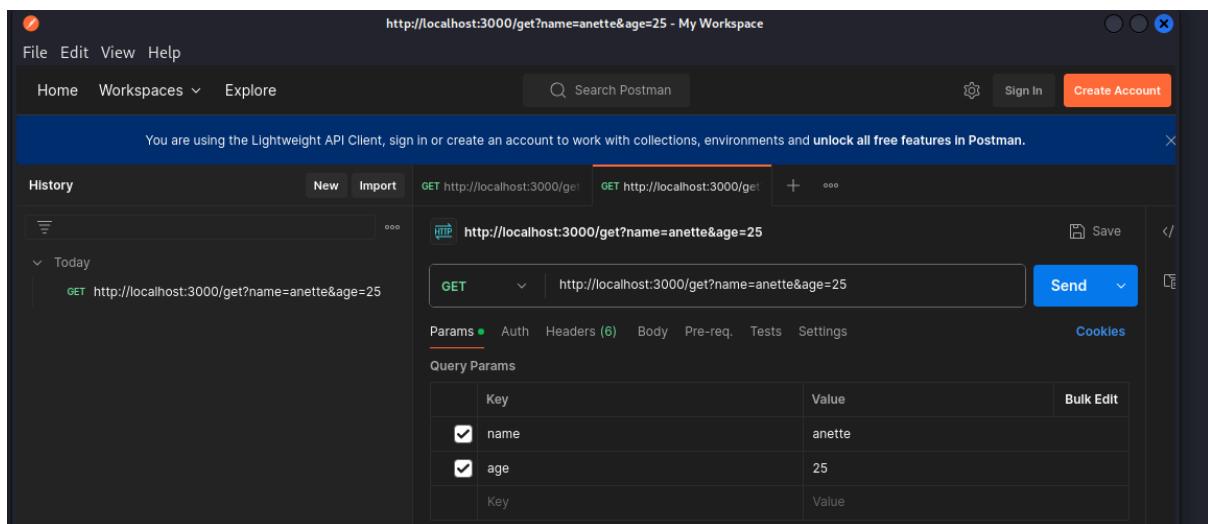
```

kali@kali: ~
File Actions Edit View Help
localhost:3000/get?name=us... + kali@kali: ~
-(kali㉿kali)-[~]
$ nano servidor_http14.js
-(kali㉿kali)-[~]
$ node servidor_http14.js
Servidor escuchando en el puerto 3000
[1] 11878 Petición GET recibida

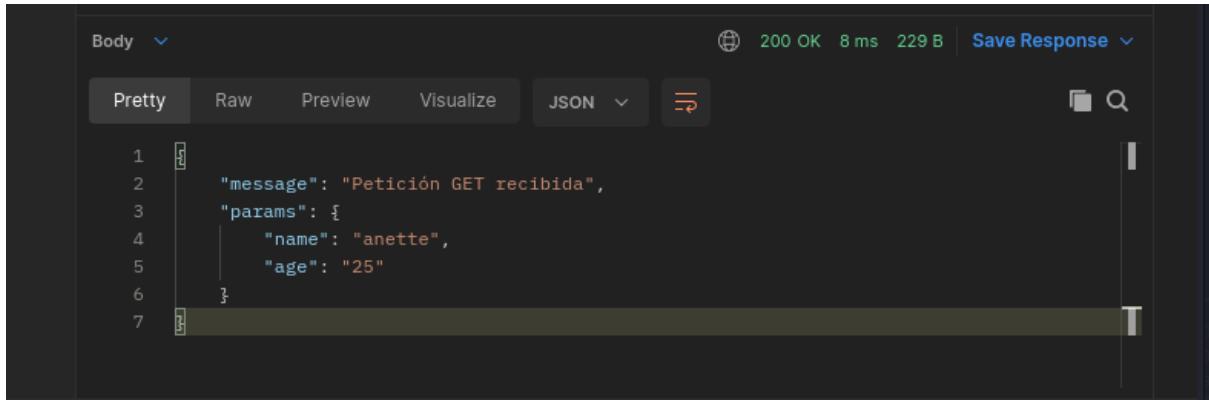
```

Como pruebas se va a utilizar el navegador y POSTMAN:

Primero en el navegador probamos para el método GET se mandan dos parámetros: un nombre y una edad (`http://localhost:3000/get?name=anette&age=25`).



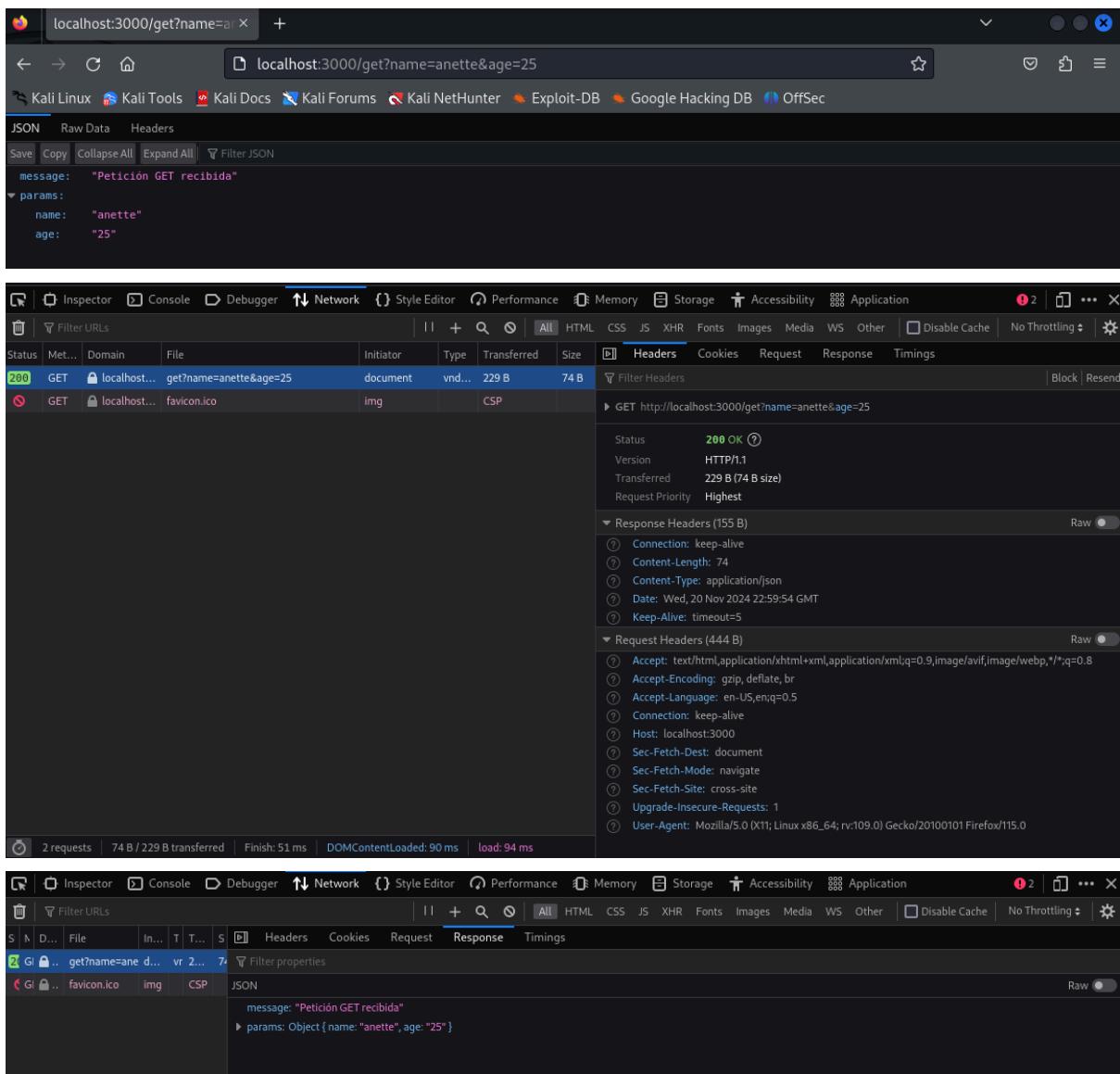
Obteniendo la siguiente respuesta, en postman:



The screenshot shows the Postman interface with a successful response. The status bar at the top indicates "200 OK" with a response time of "8 ms" and a size of "229 B". Below the status bar, there are tabs for "Pretty", "Raw", "Preview", "Visualize", and "JSON". The "JSON" tab is selected, displaying the following JSON response:

```
1
2     "message": "Petición GET recibida",
3     "params": {
4         "name": "anette",
5         "age": "25"
6     }
7
```

Ahora si se introduce la misma URL dentro del navegador y abrimos el inspector en el apartado de Redes o Network podremos ver la respuesta como se ve a continuación:



The image contains three screenshots of browser developer tools' Network tab, showing the same request and response details.

- Screenshot 1 (Top):** Shows the JSON response body with the message "Petición GET recibida" and parameters { "name": "anette", "age": "25" }.
- Screenshot 2 (Middle):** Shows the Network tab with a successful 200 OK response for the URL "localhost:3000/get?name=anette&age=25". The Headers section shows the following:
  - Status: 200 OK
  - Version: HTTP/1.1
  - Transferred: 229 B (74 B size)
  - Request Priority: HighestThe Response Headers section includes:
  - Connection: keep-alive
  - Content-Length: 74
  - Content-Type: application/json
  - Date: Wed, 20 Nov 2024 22:59:54 GMT
  - Keep-Alive: timeout=5The Request Headers section includes:
  - Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,\*/\*;q=0.8
  - Accept-Encoding: gzip, deflate, br
  - Accept-Language: en-US,en;q=0.5
  - Connection: keep-alive
  - Host: localhost:3000
  - Sec-Fetch-Dest: document
  - Sec-Fetch-Mode: navigate
  - Sec-Fetch-Site: cross-site
  - Upgrade-Insecure-Requests: 1
  - User-Agent: Mozilla/5.0 (X11; Linux x86\_64; rv:109.0) Gecko/20100101 Firefox/115.0
- Screenshot 3 (Bottom):** Shows the Network tab with a successful 200 OK response for the URL "localhost:3000/get?name=anette&age=25". The Response section shows the JSON content:

```
message: "Petición GET recibida"
params: Object { name: "anette", age: "25" }
```

Para el método POST se también se manda a la URL (`http://localhost:3000/post`) dos parámetros:

```
{  
  "name": "anette",  
  "age": 25  
}
```

The screenshot shows the Postman application interface. In the center, there is a request card for a POST method to `http://localhost:3000/post`. The 'Body' tab is selected, showing a JSON payload:

```
1  {  
2     "name": "anette",  
3     "age": 25  
4 }
```

The response section shows a 200 OK status with a duration of 484 ms and a size of 226 B. The response body is displayed as:

```
1  {  
2     "message": "Petición POST recibida",  
3     "data": {  
4         "name": "anette",  
5         "age": 25  
6     }  
7 }
```

This screenshot shows another instance of the Postman interface. It has a similar layout to the first one, with a request card for a POST method to `http://localhost:3000/post`. The 'Body' tab is selected, showing the same JSON payload as the previous screenshot:

```
1  {  
2     "name": "anette",  
3     "age": 25  
4 }
```

The response section shows a 200 OK status with a duration of 974 ms and a size of 226 B. The response body is identical to the one in the first screenshot.

This screenshot shows a Firefox browser window with the address bar set to `localhost:3000/post`. The page content displays the message: "Ruta no encontrada".

## CONCLUSIONES

Navarrete Becerril Sharon Anette:

La implementación de un servidor HTTP básico en Node.js que maneje solicitudes GET y POST demuestra cómo se pueden construir aplicaciones web sencillas pero efectivas, permitiendo la comunicación entre el cliente y el servidor a través de métodos HTTP fundamentales.

## BIBLIOGRAFÍA

Stevens, W. Richard. Programación en red con Unix: La API de Sockets. Addison-Wesley Professional, 2003.

Forouzan, Behrouz A. Comunicaciones de Datos y Redes. McGraw-Hill, 2012. Tanenbaum, Andrew S., y David J. Wetherall. Redes de Computadoras. Pearson, 2010.

Comer, Douglas E. Internetworking con TCP/IP Volumen Uno. Prentice Hall, 2006. Kurose, James F., y Keith W. Ross. Redes de Computadoras: Un Enfoque Descendente. Pearson, 2017.

Beazley, David, y Brian K. Jones. Python Cookbook: Recetas para Dominar Python 3. O'Reilly Media, 2013. Begg, A. Sockets TCP/IP en C: Guía Práctica para Programadores. Morgan Kaufmann, 2000