



# INSTITUTO POLITÉCNICO NACIONAL

## ESCUELA SUPERIOR DE CÓMPUTO



### -----APLICACIONES EN COMUNICACIONES EN RED-----

#### **PRÁCTICA 6:**

Servidor HTTP no bloqueantes

#### **Alumno:**

Meza Vargas Brandon David

#### **Grupo:**

3CM16

#### **Profesor:**

Moreno Cervantes Axel Ernesto

## Índice

<b>Introducción</b> .....	4
<b>Desarrollo</b> .....	5
<b>Server</b> .....	5
<b>initServer</b> .....	5
<b>HTTPServer</b> .....	6
<b>connection</b> .....	6
<b>getFile</b> .....	9
<b>sendFileHead</b> .....	10
<b>sendFile</b> .....	11
<b>delete</b> .....	13
<b>put</b> .....	13
<b>response</b> .....	14
<b>Constants</b> .....	15
<b>Pruebas de funcionamiento</b> .....	15
<b>Probando método GET</b> .....	15
<b>Probando método HEAD</b> .....	18
<b>Probando método POST</b> .....	19
<b>Probando método PUT</b> .....	20
<b>Probando método DELETE</b> .....	22
<b>Conclusiones</b> .....	24
<b>Bibliografía</b> .....	25

## Índice de ilustraciones

Ilustración 1. Método initServer()	6
Ilustración 2. Recibimiento de línea y método HEAD.	7
Ilustración 3. Método POST	7
Ilustración 4. Petición PUT	8
Ilustración 5. Petición delete.	8
Ilustración 6. Petición GET	9
Ilustración 7. Método getFile	10
Ilustración 8. Método sendFileHead	11
Ilustración 9. Método sendFile.	12
Ilustración 10. Método delete.	13
Ilustración 11. Método put.	14
Ilustración 12. Método response.	14
Ilustración 13. Clase Constants.	15
Ilustración 14. Corriendo server.	15
Ilustración 15. Formulario	16
Ilustración 16. Formulario lleno.	16
Ilustración 17. Parámetros obtenidos usando GET	16
Ilustración 18. Respuesta servidor	17
Ilustración 19. Pidiendo una imagen.	18
Ilustración 20. Respuesta del servidor	18
Ilustración 21. Petición HEAD desde postman	18
Ilustración 22. Respuesta HEAD	19
Ilustración 23. Formulario para método post.	19
Ilustración 24. Obteniendo parámetros.	19
Ilustración 25. Respuesta servidor.	20
Ilustración 26. PUT con postman	20
Ilustración 27. Respuesta servidor.	21
Ilustración 28. Respuesta de postman	21
Ilustración 29. Actualizando archivo	21
Ilustración 30. Respuesta del servidor.	22
Ilustración 31. Respuesta de postman	22
Ilustración 32. Archivo borrado con postman.	23
Ilustración 33. Respuesta servidor método DELETE.	23

## Introducción

Un servidor web es un software que forma parte del servidor y tiene como misión principal devolver información cuando recibe peticiones por parte de los usuarios. Para que un servidor web funcione correctamente necesitamos un cliente web que realice una petición http a través de un navegador como Chrome, Firefox o Safari y un servidor donde esté almacenada la información, también estas peticiones se pueden realizar usando distintas herramientas, entre ellas Postman, la cual usamos en esta práctica para probar las peticiones, HEAD, PUT y DELETE.

La comunicación entre un servidor y sus clientes se basa en HTTP y la principal función del servidor es mostrar el contenido de un sitio web almacenando, procesando y entregando las páginas web a los usuarios.

En la presente práctica se realizará un servidor HTTP, con el que podamos interactuar con clientes que hagan peticiones HTTP, entre ellas se implementarán los métodos GET, POST, HEAD, DELETE y PUT, todo esto usando sockets no bloqueantes.

Los sockets en modo bloqueante, las llamadas de eventos del sistema se esperan hasta que una respuesta apropiada llegue, los no bloqueantes, a diferencia de estos, continúan con su ejecución incluso si las llamadas al sistema han sido invocadas y trata con ellas de manera apropiada más tarde incluso si las llamadas no han sido completadas y cada llamada se trata de manera separada, por esta razón, usando este tipo de sockets no es necesario el uso de hilos.

## Desarrollo

A continuación, se muestra el desarrollo de la práctica con las capturas de todo el código utilizado

### Server

Esta es la clase que inicia el servidor usando sockets no bloqueantes, dentro de esta clase tenemos un método llamado `initServer` donde se inicia la conexión.

#### `initServer`

Primeramente, se crea un selector que es un multiplexor para el objeto `SelectableChannel`, un selector se crea invocando a su método `open`, la cual usa el selector por defecto del sistema para crear uno nuevo. Posteriormente creamos un canal usando la clase `ServerSocketChannel` y su método `open()`, una instancia de esta clase la usamos para establecer una conexión entre el cliente y el servidor como la clase `Socket`, después configuramos este canal como no bloqueante y hacemos el `bind` con un puerto establecido y registramos el canal creado con el selector igualmente creado, retornándonos una llave de selección.

Después dentro de un ciclo infinito donde seleccionamos un grupo de llaves correspondientes a los canales listos para operaciones de E/S.

Si la llave es aceptable, haremos la aceptación del cliente y configuramos al cliente como no bloqueante al igual que el servidor. Si la llave puede ser leída es donde lanzamos nuestro servidor HTTP.

```
private void initServer(){
    try {
        Selector selector = Selector.open();
        s = ServerSocketChannel.open();
        s.configureBlocking(false);
        s.setOption(StandardSocketOptions.SO_REUSEADDR, true);
        s.socket().bind(new InetSocketAddress(port));
        s.register(selector, SelectionKey.OP_ACCEPT);
        System.out.println(Constants.SERVER_CONNECTION_START);
        while(true){
            selector.select();
            Iterator<SelectionKey> it = selector.selectedKeys().iterator();
            while(it.hasNext()){
                SelectionKey k = (SelectionKey) it.next();
                it.remove();
                if(k.isAcceptable()){
                    SocketChannel cl = s.accept();
                    System.out.println(String.format(Constants.CONNECTED_CLIENT_MESSAGE, cl.socket().getInetAddress().getHostAddress()));
                    cl.configureBlocking(false);
                    cl.register(selector, SelectionKey.OP_READ);
                    continue;
                }
            }
        }
    }
}
```

```

        if(k.isReadable()){
            SocketChannel ch = (SocketChannel) k.channel();
            HTTPServer httpServer = new HTTPServer(ch);
            httpServer.connection();
        }
    }
}
} catch (IOException e){
    throw new RuntimeException(String.format(Constants.SERVER_CONNECTION_ERROR, s.socket().getLocalPort()), e);
}
}

```

Ilustración 1. Método initServer()

## HTTPServer

En el servidor tenemos toda la lógica implementada de los métodos HEAD, GET, POST, DELETE y PUT.

### connection

Primeramente, establecemos nuestro ByteBuffer para asignarle memoria, posteriormente leemos lo que nos viene del socket, esto será nuestra petición HTTP, en caso de que sea nula indicamos que nos llegó una petición vacía. Para comenzar a estructurar nuestra respuesta lo token izamos a partir de cada salto de línea, esta línea nos indicará el tipo de petición http que será.

Primeramente, preguntamos si la línea viene con parámetros, en caso de que no podremos descartar que se trató de una petición GET ya que en esta se mandan parámetros visibles separados por un '?'. En primer lugar, tenemos al método HEAD, este es un método idéntico a GET pero sin el cuerpo de la petición, por lo que solo mandamos los headers con la información del archivo, si no se indica el archivo se mandará por defecto un archivo index.html.

```

public void connection(){
    try {
        b = ByteBuffer.allocate(Constants.INPUT_STREAM_BYTES_SIZE);
        b.clear();
        int t = socket.read(b);
        b.flip();
        String request = new String(b.array(), offset 0,t);
        System.out.println("t: " + t);
        if(request == null){
            StringBuilder sb = new StringBuilder();
            sb.append("<html><head><title>Servidor WEB\n");
            sb.append("</title><body bgcolor=#AACCFF\n");
            sb.append("<br>Linea Vacía\n");
            sb.append("</body></html>\n");
            dos.write(sb.toString().getBytes());
            dos.flush();
            socket.close();
            return;
        }
        System.out.println(String.format(Constants.CONNECTED_CLIENT_MESSAGE,socket.socket().getInetAddress(), socket.socket().getInetAddress().getHostAddress()));
        System.out.println(String.format(Constants.CONNECTED_CLIENT_DATA, request));
        StringTokenizer stl = new StringTokenizer(request, delim: "\n");
        String line = stl.nextToken();
    }
}

```

```

String line = sc.nextLine();
if(line.indexOf("?") == -1){
    if(line.toUpperCase().startsWith("HEAD")){
        getFile(line);
        if(fileName.compareTo("") == 0)
            sendFileHead( arg: "index.htm", dos);
        else sendFileHead(fileName,dos);
    }
}

```

*Ilustración 2. Recibimiento de línea y método HEAD.*

Si la línea nos indica que se trata de un método POST lo que hacemos es encontrar los parámetros que se encuentran al final de la línea, una vez esto mandamos la respuesta con los mensajes apropiados e indicando los parámetros obtenidos, como se trata de una petición POST, estos parámetros no se visualizan en la url del navegador, pero si los mostramos en pantalla para ver que se obtuvieron los parámetros correctos.

```

else if(line.toUpperCase().startsWith("POST")){
    String params = request.substring(request.lastIndexOf( str: "\n"));
    /**Getting response
    StringBuffer response = new StringBuffer();
    response.append("HTTP/1.0 200 OK \n");
    String date= "Date: " + new Date()+" \n";
    response.append(date);
    String tipo_mime = "Content-Type: text/html \n\n";
    response.append(tipo_mime);
    response.append("<html><head><title>SERVIDOR WEB</title></head>\n");
    response.append("<body bgcolor=\"#AACCFF\"><center><h1><br> Parametros obtenidos </br></h1><h3><b>\n");
    response.append(params);
    response.append("</b></h3>\n");
    response.append("</center></body></html>\n\n");
    System.out.println("Respuesta: "+response);
    b = ByteBuffer.wrap(response.toString().getBytes());
    socket.write(b);
    socket.close();
}

```

*Ilustración 3. Método POST*

En caso de ser una petición de tipo PUT, mandamos el nombre del archivo a ser modificado y mandamos a llamar al método put explicado más adelante.

```

}else if(line.toUpperCase().startsWith("PUT")){
    getFile(line);
    while(!line.contains("Content-Type"))
        line = stl.nextToken();
    stl.nextToken();
    put(stl);
}

```

*Ilustración 4. Petición PUT*

Finalmente, en caso de tratarse de una petición DELETE, se obtiene el nombre del archivo y se manda a llamar el método delete explicado más adelante.

```

}else if(line.toUpperCase().startsWith("DELETE")){
    getFile(line);
    delete();
}

```

*Ilustración 5. Petición delete.*

En caso de que la línea venga con parámetros en el url, sabremos que se trata de una petición GET, aquí lo que hacemos es obtener el nombre del archivo en caso de que se indique, si no se indica se manda el archivo index.htm por defecto, posteriormente obtenemos los parámetros sabiendo que están separados por un '?', a partir de esto preparamos la respuesta que daremos indicando los parámetros obtenidos y los escribiéndolo en el socket obteniendo los bytes de la respuesta.

En caso de recibir una petición no implementada se mandará un mensaje indicando que no se ha implementado.



```

}else if(line.toUpperCase().startsWith("GET")){
    /*
    getFile(line);
    if(fileName.compareTo("") == 0)
        sendFile("index.htm", dos);
    else sendFile(fileName,dos);

    StringTokenizer tokens = new StringTokenizer(line, " ");
    String req_a = tokens.nextToken();
    String req = tokens.nextToken();
    System.out.println("Token1: " + req_a);
    System.out.println("Token2: " + req);
    String params = req.substring(0, req.indexOf(" "))+ "\n";
    System.out.println("Params: " + params);
    /*Getting response
    StringBuffer response = new StringBuffer();
    response.append("HTTP/1.0 200 OK \n");
    String date= "Date: " + new Date()+" \n";
    response.append(date);
    String tipo_mime = "Content-Type: text/html \n\n";
    response.append(tipo_mime);
    response.append("<html><head><title>SERVIDOR WEB</title></head>\n");
    response.append("<body bgcolor=\"#AACCFF\"><center><h1><br> Parametros obtenidos </br></h1><h3><b>\n");
    response.append(params);
    response.append("</b></h3>\n");
    response.append("</center></body></html>\n\n");
    System.out.println("Respuesta: "+response);
    b = ByteBuffer.wrap(response.toString().getBytes());
    socket.write(b);
    socket.close();
    }else{
        b = ByteBuffer.wrap(Constants.HTTP_NOT_IMPLEMENTED.getBytes());
        socket.write(b);
        socket.close();
    }
}catch(IOException e){
    e.printStackTrace();
}

```

Ilustración 6. Petición GET

## getFile

Este método solo se encarga de obtener el nombre del archivo solicitado y determinar la extensión de este, la extensión nos servirá para determinar el content-type de las peticiones.

```

/**
 * Method that obtains the file name and its extension
 * @param line, the line received
 */
public void getFile(String line){
    int i, f;
    i = line.indexOf("/");
    f = line.indexOf(" ", i);
    fileName = line.substring(i+1, f);

    /*Getting extension
    extension = "";
    i = fileName.lastIndexOf('.');
    if(i > 0) extension = fileName.substring(i+1);
    }

```

Ilustración 7. Método getFile

## sendFileHead

En este método se encarga de mandar el archivo solicitado cuando es una petición HEAD, recibe el nombre del archivo y se crea el cuerpo, de acuerdo con la extensión se determina el Content-Type correcto y se junta con el cuerpo para ser enviado, en este caso el cuerpo son los headers ya que se trata de una petición HEAD.

```

public void sendFileHead(String arg, DataOutputStream dos1){
    try{

        String extension = "";
        int i = arg.lastIndexOf('.');
        if(i > 0) extension = arg.substring(i+1);

        System.out.println("Extension "+ extension);
        DataInputStream dis2 = new DataInputStream(new FileInputStream(arg));
        File ff = new File(arg);
        long fileSize = ff.length();
        /******
        String sb = "";
        sb = sb+"HTTP/1.0 200 ok\n";
        sb = sb + "Server: BrandonMV Server/1.0 \n";
        sb = sb + "Date: " + new Date()+" \n";
        switch (extension){
            case "htm":
            case "html":
                sb = sb + "Content-Type: text/html \n";
                break;

```

```

        case "jpg":
        case "jpeg":
            sb = sb + "Content-Type: image/jpeg \n";
            break;
        case "pdf":
            sb = sb + "Content-Type: application/pdf \n";
            break;
    }
    //sb = sb + "Content-Type: text/html \n";
    sb = sb + "Content-Length: "+fileSize+" \n";
    sb = sb + "\n";
    b = ByteBuffer.wrap(sb.getBytes());
    socket.write(b);
    /***/
    dis2.close();
    socket.close();
} catch (IOException e){
    System.err.println(e.getMessage());
}

```

*Ilustración 8. Método sendFileHead*

## sendFile

Este método es lo mismo que lo anterior, pero aquí si mandaremos el archivo solicitado ya que se tratará de una petición GET, este archivo se envía usando un ciclo while donde se van leyendo ls bytes del archivo y se van escribiendo en el flujo de salida.

```

public void sendFile(String arg, DataOutputStream dos1){
    try{
        String extension = "";
        int i = arg.lastIndexOf( ch: '.');
        if(i > 0) extension = arg.substring( beginIndex: i+1);

        DataInputStream dis2 = new DataInputStream(new FileInputStream(arg));
        byte[] buf = new byte[Constants.INPUT_STREAM_BYTES_SIZE];
        int x = 0;
        File ff = new File(arg);
        long fileSize = ff.length(), cont = 0;
        /*******
        String sb = "";
        sb = sb+"HTTP/1.0 200 ok\n";
        sb = sb + "Server: BrandonMV Server/1.0 \n";
        sb = sb + "Date: " + new Date()+" \n";
        switch (extension){
            case "htm":
            case "html":
                sb = sb + "Content-Type: text/html \n";
                break;
            case "jpg":
                sb = sb + "Content-Type: image/jpeg \n";
                break;
            case "pdf":
                sb = sb + "Content-Type: application/pdf \n";
                break;
        }
        //sb = sb + "Content-Type: text/html \n";
        sb = sb + "Content-Length: "+fileSize+" \n";
        sb = sb + "\n";
        b = ByteBuffer.wrap(sb.getBytes());
        socket.write(b);
        /*******
        while(cont < fileSize){
            x = dis2.read(buf);
            b = ByteBuffer.wrap(buf, offset: 0 , x);
            socket.write(b);
            cont+=x;
        }
        dis2.close();
        socket.close();
    }catch (IOException e){
        System.err.println(e.getMessage());
    }
}

```

Ilustración 9. Método sendFile

## delete

Esté método realiza la petición delete, solo se encarga de borrar el archivo indicado en la petición, posteriormente se manda la respuesta dependiendo si fue posible borrar el archivo, si no se pudo borrar o si no se encontró el archivo.

```
/**
 * The delete HTTP method, deletes the indicated file
 * @throws IOException
 */
private void delete() throws IOException{
    File f = new File(fileName);
    if(f.exists()){
        if(f.canWrite()){
            f.delete();
            /**Getting response
            response( statusCode: "200", meaning: "OK", msg: "File deleted");
            }else{
                response( statusCode: "403", meaning: "Forbidden", msg: "403 FORBIDDEN");
            }
        }else{
            response( statusCode: "404", meaning: "Not Found", msg: "404 Not Found");
        }
    }
}
```

*Ilustración 10. Método delete.*

## put

Este método se encarga de realizar la petición PUT, si el archivo existe esto quiere decir que se editará el archivo, para esto se elimina el archivo que se editará y se guardará el mismo archivo, pero con los cambios realizados, en caso de que el archivo no exista se creará uno nuevo. Cada operación manda la respuesta correspondiente de creación o edición de archivo.

```

/**
 * Method that updates or creates a file
 * @param stl The file content
 * @throws IOException Excpetion thrown
 */
private void put(StringTokenizer stl) throws IOException{
    FileWriter fw;
    File f = new File(fileName); /* File to create or update
    File f2 = new File(fileName); /* File updated
    /*If the file exists and is writable means that the file will be updated
    if(f.exists()){
        if(f.canWrite()){
            f.delete();
            f2.createNewFile();
            fw = new FileWriter(fileName);
            while(stl.hasMoreTokens())
                fw.write(stl.nextToken());
            fw.close();
            response( statusCode: "200", meaning: "OK", msg: "Archivo actualizado");
        }else response( statusCode: "403", meaning: "Forbidden", msg: "403 Forbidden");/* Sin permiso para escribir*/
    }else{
        /* If file does not exist we have to create it
        f.createNewFile();
        response( statusCode: "201", meaning: "Created", msg: "Archivo creado");
        fw = new FileWriter(fileName);
        while(stl.hasMoreTokens())
            fw.write(stl.nextToken());
        fw.close();
    }
}

```

Ilustración 11. Método put.

## response

Este método se encarga de estructurar la respuesta HTTP, indicando las cabeceras de la respuesta y un cuerpo que contiene un archivo html que mostrará un mensaje dependiendo de la petición que se haga.

```

private void response(String statusCode, String meaning, String msg) throws IOException{
    StringBuffer response = new StringBuffer();
    response.append("HTTP/1.0 " + statusCode + meaning + " \n");
    String date= "Date: " + new Date()+" \n";
    response.append(date);
    String tipo_mime = "Content-Type: text/html \n\n";
    response.append(tipo_mime);
    response.append("<html><head><title>SERVIDOR WEB</title></head>\n");
    response.append("<body bgcolor=\"#AACCFF\"><center><h1><br>" + msg + "</br></h1><h3><b>\n");
    response.append("</b></h3>\n");
    response.append("</center></body></html>\n\n");
    System.out.println("Respuesta: "+response);
    b = ByteBuffer.wrap(response.toString().getBytes());
    socket.write(b);
    socket.close();
}

```

Ilustración 12. Método response.

## Constants

En esta clase solo se guardan los mensajes y algunas configuraciones usadas en todo el programa como constantes.

```
public static final int PORT = 8000;
//public static final int THREADS = 4;

/**Server messages
1 usage
public static final String SERVER_CONNECTION_ERROR = "No se pudo iniciar el socket en el puerto %d";
1 usage
public static final String SERVER_CONNECTION_START = "Servidor iniciado, esperando clientes...";
public static final String SERVER_CONNECTION_ACCEPTED = "Conexion aceptada...";
public static final String SERVER_CONNECTION_STOPPED = "Servidor detenido";
public static final String SERVER_CONNECTION_ACCEPTED_ERROR = "Error al aceptar una nueva conexion";
public static final String SERVER_CONNECTION_STOPPED_ERROR = "Error al cerrar el socket del servidor";

/**Utilities
2 usages
public static final int INPUT_STREAM_BYTES_SIZE = 1024;
2 usages
public static final String CONNECTED_CLIENT_MESSAGE = "\nCliente conectado desde: %s\nPor el puerto: %d";
1 usage
public static final String CONNECTED_CLIENT_DATA = "Datos: %s\r\n\r\n";

1 usage
public static final String HTTP_NOT_IMPLEMENTED = "HTTP/1.0 501 Not Implemented\r\n";
```

*Ilustración 13. Clase Constants.*

## Pruebas de funcionamiento

Primeramente, tenemos que correr el server usando la clase Server y estará a la espera de clientes como se muestra en la siguiente ilustración.

```
Servidor iniciado, esperando clientes...
```

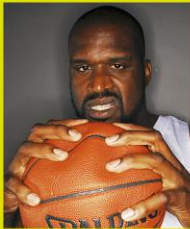
*Ilustración 14. Corriendo server*

## Probando método GET

Primeramente, probaremos el método get, para esto tenemos una página con dos formularios, uno corresponde al método get y el otro al método post.

Ejemplo sencillo de un formulario

Formulario de Registro



Sea tan amable de rellenar el siguiente formulario.

Nombre:

Dirección:

Teléfono:

Comentarios:

Ilustración 15. Formulario

Ahora bien, para probar el método se llenará el formulario y se enviará para recibir la respuesta con los parámetros.

Sea tan amable de rellenar el siguiente formulario.

Nombre:

Dirección:

Teléfono:

Comentarios:

Ilustración 16. Formulario lleno.

Una vez lleno y enviado recibimos la respuesta que se ve en la siguiente ilustración.

**Parametros obtenidos**

Apellido=brandon&Direccion=Ecatepec&Telefono=5610690179&comentario=Hola+estamos+probando+el+metodo+get

Ilustración 17. Parámetros obtenidos usando GET



La respuesta que obtenemos en nuestro servidor es la siguiente

```
Cliente conectado desde: /127.0.0.1
Por el puerto: 54890
Datos: GET /?Apellido=brandon&Direccion=Ecatepec&Telefono=5610690179&comentario=Hola+estamos+probando+el+metodo+get HTTP/1.1
Host: 127.0.0.1:8000
Connection: keep-alive
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="100", "Microsoft Edge";v="100"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36 Edg/100.0.1185.44
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: es-419,es;q=0.9,es-ES;q=0.8,en;q=0.7,en-GB;q=0.6,en-US;q=0.5

Token1: GET /
Token2: Apellido=brandon&Direccion=Ecatepec&Telefono=5610690179&comentario=Hola+estamos+probando+el+metodo+get HTTP/1.1
Params: Apellido=brandon&Direccion=Ecatepec&Telefono=5610690179&comentario=Hola+estamos+probando+el+metodo+get

Respuesta: HTTP/1.0 200 OK
Date: Fri Apr 22 15:19:27 CDT 2022
Content-Type: text/html

<html><head><title>SERVIDOR WEB</title></head>
<body bgcolor="#AACCFF"><center><h1><br> Parametros obtenidos </br></h1><h3><b>
Apellido=brandon&Direccion=Ecatepec&Telefono=5610690179&comentario=Hola+estamos+probando+el+metodo+get
</b></h3>
</center></body></html>
```

*Ilustración 18. Respuesta servidor*

De igual forma podemos obtener archivos colocando el nombre en el url.

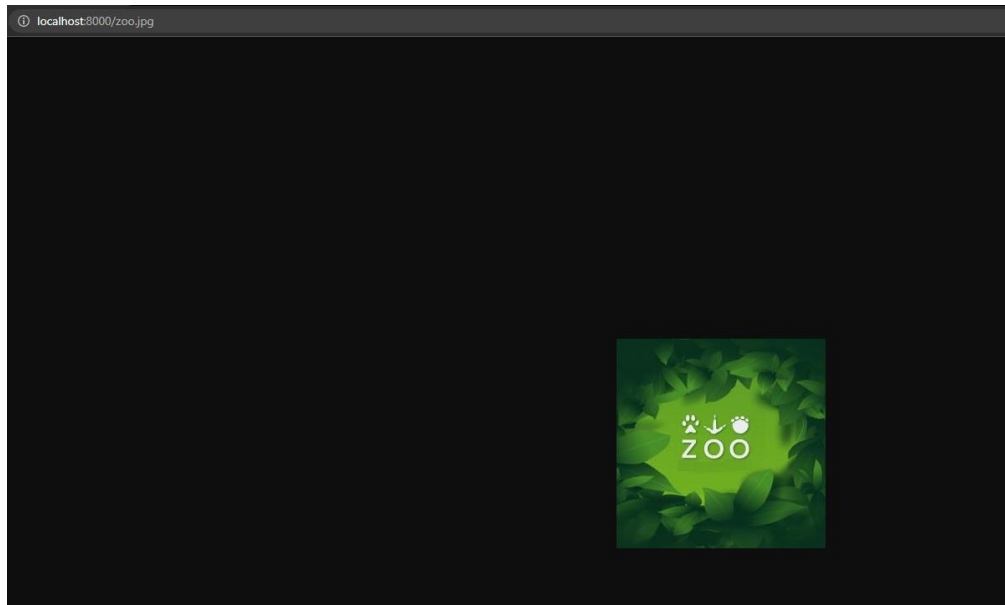


Ilustración 19. Pidiendo una imagen

En el servidor tenemos la siguiente respuesta.

```
Conexion aceptada...
t: 671

Cliente conectado desde: /0:0:0:0:0:0:1
Por el puerto: 51895
Datos: GET /zoo.jpg HTTP/1.1
Host: localhost:8000
Connection: keep-alive
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="100", "Google Chrome";v="100"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: es-ES,es;q=0.9
```

Ilustración 20. Respuesta del servidor

## Probando método HEAD

Ahora veremos la petición HEAD, para esto nos ayudaremos de postman como se ve a continuación.

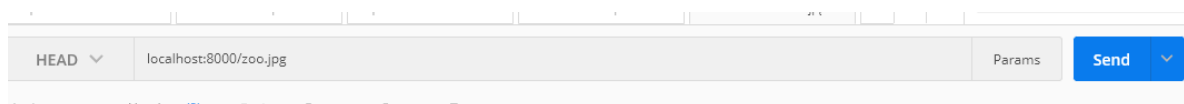


Ilustración 21. Petición HEAD desde postman

En el servidor obtenemos lo siguiente

```
Conexion aceptada...
t: 561

Cliente conectado desde: /0:0:0:0:0:0:1
Por el puerto: 51968
Datos: HEAD /zoo.jpg HTTP/1.1
Host: localhost:8000
Connection: keep-alive
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="100", "Google Chrome";v="100"
Cache-Control: no-cache
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36
sec-ch-ua-platform: "Windows"
Postman-Token: c0eee910-350e-0a4e-f2e9-3a3de8b8fa5a
Accept: */*
Sec-Fetch-Site: none
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Accept-Encoding: gzip, deflate, br
Accept-Language: es-ES,es;q=0.9
```

Ilustración 22. Respuesta HEAD

## Probando método POST

Para este método volvemos a recurrir al formulario, mandando los datos obtenemos la siguiente respuesta.



Nombre:

Dirección:

Teléfono:

Comentarios:

Ilustración 23. Formulario para método post.

**Parametros obtenidos**

**Apellido=Brandon&Direccion=123&Telefono=5610690179&comentario=Probando+metodo+post**

Ilustración 24. Obteniendo parámetros

```

Cliente conectado desde: /127.0.0.1
Por el puerto: 52021
Datos: POST / HTTP/1.1
Host: 127.0.0.1:8000
Connection: keep-alive
Content-Length: 82
Cache-Control: max-age=0
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="100", "Google Chrome";v="100"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
Origin: http://localhost:8000
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:8000/
Accept-Encoding: gzip, deflate, br
Accept-Language: es-ES,es;q=0.9

Apellido=Brandon&Direccion=123&Telefono=5610690179&comentario=Probando+metodo+post

Respuesta: HTTP/1.0 200 OK
Date: Fri Apr 22 17:05:07 CDT 2022
Content-Type: text/html

<html><head><title>SERVIDOR WEB</title></head>
<body bgcolor="#AACCFF"><center><h1><br> Parametros obtenidos </br></h1><h3><b>
Apellido=Brandon&Direccion=123&Telefono=5610690179&comentario=Probando+metodo+post</b></h3>
</center></body></html>

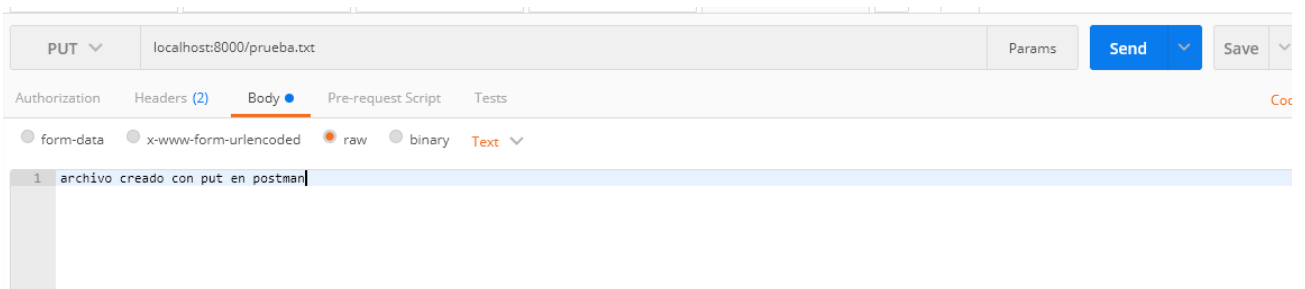
```

*Ilustración 25. Respuesta servidor.*

## Probando método PUT

De igual forma nos ayudaremos de postman para probar este método, vamos a crear un nuevo archivo y después editarlo.

Primero lo crearemos usando postman como se ve a continuación.



*Ilustración 26. PUT con postman*

Una vez creado postman y el servidor nos muestran las siguientes respuestas.

```

Cliente conectado desde: /0:0:0:0:0:0:1
Por el puerto: 52104
Datos: PUT /prueba.txt HTTP/1.1
Host: localhost:8000
Connection: keep-alive
Content-Length: 33
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="100", "Google Chrome";v="100"
Cache-Control: no-cache
Content-Type: text/plain;charset=UTF-8
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36
sec-ch-ua-platform: "Windows"
Postman-Token: eaa4fa7c-2a2c-c164-4648-c7d2265c0a6f
Accept: */*
Origin: chrome-extension://fhbjgbiflinjbdggehcdcbncdddomop
Sec-Fetch-Site: none
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Accept-Encoding: gzip, deflate, br
Accept-Language: es-ES,es;q=0.9

archivo creado con put en postman

Respuesta: HTTP/1.0 201Created
Date: Fri Apr 22 17:09:36 CDT 2022
Content-Type: text/html

<html><head><title>SERVIDOR WEB</title></head>
<body bgcolor="#AACCFF"><center><h1><br>Archivo creado</br></h1><h3><b>
</b></h3>
</center></body></html>

```

Ilustración 27. Respuesta servidor.

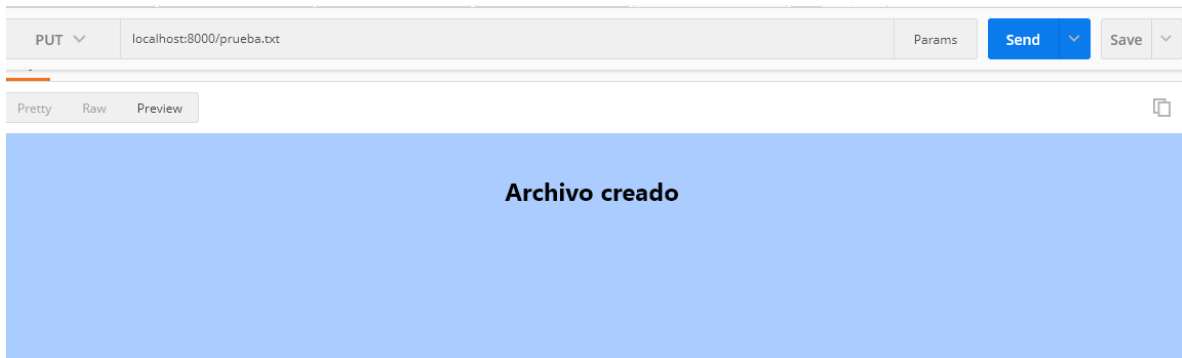


Ilustración 28. Respuesta de postman

Ahora actualizaremos ese archivo.

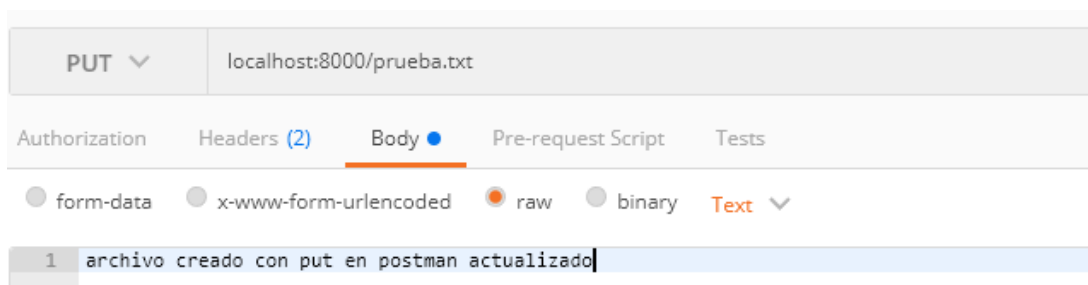


Ilustración 29. Actualizando archivo

Al actualizar el archivo se nos muestran las siguientes respuestas.

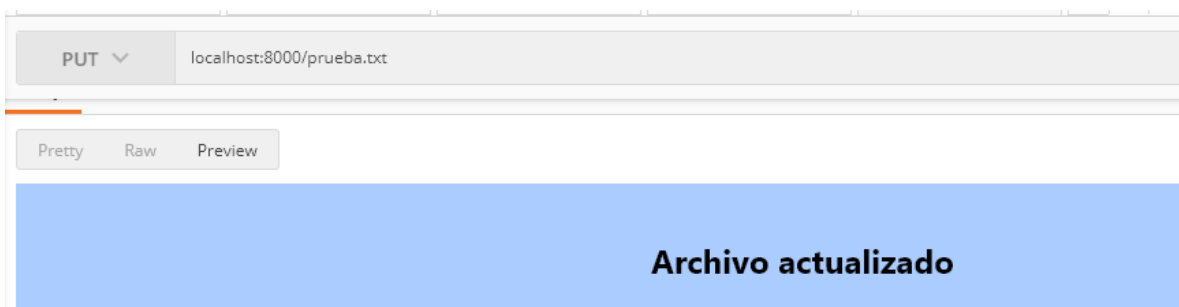
```
Cliente conectado desde: /0:0:0:0:0:0:1
Por el puerto: 52146
Datos: PUT /prueba.txt HTTP/1.1
Host: localhost:8000
Connection: keep-alive
Content-Length: 45
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="100", "Google Chrome";v="100"
Cache-Control: no-cache
Content-Type: text/plain; charset=UTF-8
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36
sec-ch-ua-platform: "Windows"
Postman-Token: a49dd994-d4e1-107a-d59e-1d5fef72533e
Accept: */*
Origin: chrome-extension://fhbjgbiflinjbdgghehcdcbncdddomop
Sec-Fetch-Site: none
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Accept-Encoding: gzip, deflate, br
Accept-Language: es-ES,es;q=0.9

archivo creado con put en postman actualizado

Respuesta: HTTP/1.0 200OK
Date: Fri Apr 22 17:12:01 CDT 2022
Content-Type: text/html

<html><head><title>SERVIDOR WEB</title></head>
<body bgcolor="#AACCFF"><center><h1><br>Archivo actualizado</br></h1><h3><b>
</b></h3>
</center></body></html>
```

*Ilustración 30. Respuesta del servidor.*



*Ilustración 31. Respuesta de postman*

## Probando método DELETE

Para este método usaremos de igual forma postman, eliminando el archivo creado con el método put anteriormente como se ve a continuación,

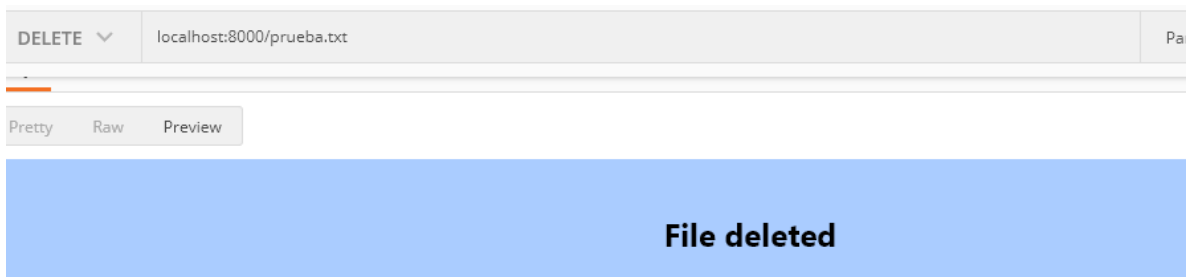


Ilustración 32. Archivo borrado con postman

La respuesta arrojada por el servidor fue la siguiente.

```
Cliente conectado desde: /0:0:0:0:0:0:1
Por el puerto: 52216
Datos: DELETE /prueba.txt HTTP/1.1
Host: localhost:8000
Connection: keep-alive
Content-Length: 45
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="100", "Google Chrome";v="100"
Cache-Control: no-cache
Content-Type: text/plain;charset=UTF-8
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36
sec-ch-ua-platform: "Windows"
Postman-Token: 01acac94-f218-f12d-5919-6b3e73ba8a6c
Accept: */*
Origin: chrome-extension://fhbjgbiflinjbdgghehcdcbncdddomop
Sec-Fetch-Site: none
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Accept-Encoding: gzip, deflate, br
Accept-Language: es-ES,es;q=0.9

Respuesta: HTTP/1.0 200OK
Date: Fri Apr 22 17:15:18 CDT 2022
Content-Type: text/html

<html><head><title>SERVIDOR WEB</title></head>
<body bgcolor="#AACCFF"><center><h1><br>File deleted</br></h1><h3><b>
</b></h3>
</center></body></html>
```

Ilustración 33. Respuesta servidor método DELETE

## **Conclusiones**

Con la realización de la práctica pude comprender de mejor manera como se usan los sockets no bloqueantes, podemos darnos cuenta de que obtenemos la misma funcionalidad que en este caso fue la práctica del servidor HTTP, con la particularidad que al estar usando sockets no bloqueantes, no tenemos la necesidad de implementar hilos como en la práctica 4 donde usamos sockets bloqueantes.



## **Bibliografía**

1. Debnath, M, (2018). "What is non-blobking socket programming in Java". Obtenido de: <https://www.developer.com/java/data/what-is-non-blocking-socket-programming-in-java/>