

TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- Conceptos Rest.
- Concepto de rutas en contexto de servidor.
- Verbos HTTP.

API REST

Hasta este punto del curso, ya hemos hablado en varias ocasiones respecto al concepto **API**. Ahora, repasaremos un concepto que suele ir fuertemente relacionado cuando hablamos de éste. Es el concepto **Rest**, que significa Representational State Transfer, y se refiere a un estilo de arquitectura para la construcción de programas, basados en la comunicación por medio del protocolo **HTTP**.

Una **API Rest** ofrece puntos de acceso o rutas al programa, a las cuales se accede mediante peticiones **HTTP**, utilizando los diferentes verbos que este protocolo ofrece.

VERBOS HTTP

Los principales son: **GET, POST, PUT, DELETE**. Estos verbos se relacionan directamente al concepto **CRUD**.

Cada vez que ingresas a una dirección URL dentro de tu navegador, estás realizando una petición **HTTP** a un servidor, el cual se encarga de procesarla, y devolver una respuesta. En estos casos, generalmente estamos hablando de una petición de tipo **GET**, con la cual solo se pretende acceder a la información.

Cuando llenas un formulario en una página web, ya sea para crear una nueva cuenta o responder una encuesta, al momento de enviar los datos, estás haciendo una petición de tipo **POST**. Las peticiones de tipo post siempre llevan información con ellas, y son insertadas en una base de datos, dependiendo del tipo de **API Rest**.

Cuando estás actualizando datos, o cuando olvidas tu contraseña, por ejemplo, y pides su cambio, es muy probable que estés realizando una petición de tipo **PUT**, donde actualizas información ya existente en la base de datos.

Y, por último, cuando eliminas tu cuenta de alguna página web, estás realizando una petición de tipo **DELETE**, en donde se eliminan los datos existentes en la base de datos.

PETICIONES HTTP DENTRO DE NODE

Anteriormente, ya hemos hablado sobre cómo crear un servidor simple utilizando **node**.

```
JS index.js X
JS index.js > ...
1  const http = require('http');
2
3  http.createServer(function (req, res) {
4    res.write("Respuesta desde servidor node !");
5    res.end();
6  })
7  .listen(3000, function(){
8    console.log("Servidor iniciado en puerto 3000");
9  });
10
11
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\nodeporcomandos>node index.js
Servidor iniciado en puerto 3000
█
```

Ahora, hablaremos sobre **request** y **response**. Estos argumentos generalmente los verás escritos de forma abreviada.

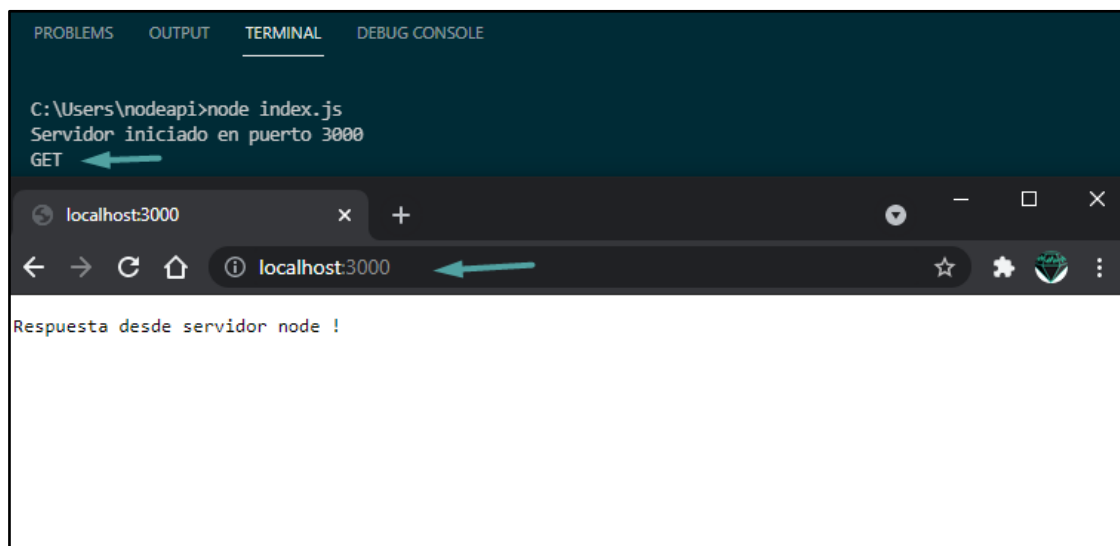
```
2
3  http.createServer(function (req, res) {
4    res.write("Respuesta desde servidor node !");
5    res.end();
6  })
```

Request es un objeto que tiene una gran cantidad de datos, referentes a la petición realizada desde el navegador. Dentro de éste, encontraremos dos propiedades que nos serán de mucha utilidad.

Primero, revisemos la propiedad método.

```
2
3 http.createServer(function (req, res) {
4     console.log(req.method);
5
6     res.write("Respuesta desde servidor node !");
7     res.end();
8 }
9 )
10 .listen(3000, function(){
11     console.log("Servidor iniciado en puerto 3000");
12 });
13
```

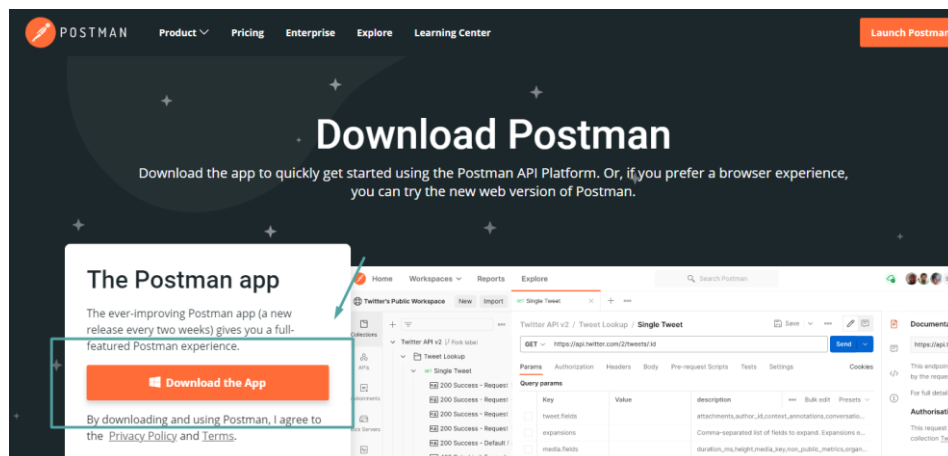
Ésta contiene la información respecto al verbo con el cual se está enviando la petición HTTP. Si hacemos una llamada desde nuestro navegador a la URL "localhost:3000", podremos ver que el método usado es de tipo **GET**.




POSTMAN

Haremos un pequeño desvío, para hablar sobre un programa que nos será de gran ayuda para poder probar la funcionalidad de nuestro servidor.

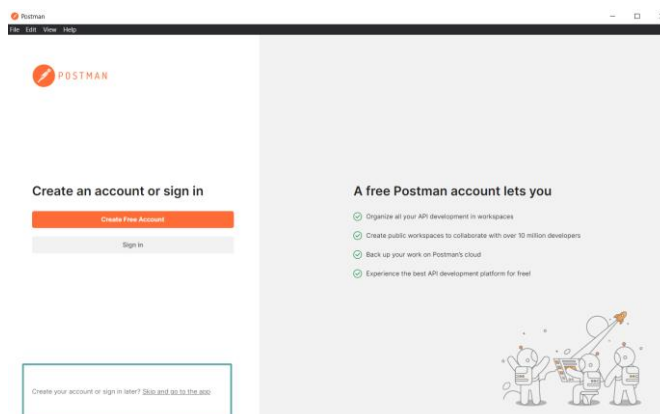
Descargaremos **postman** desde la siguiente página: <https://www.postman.com/downloads/>



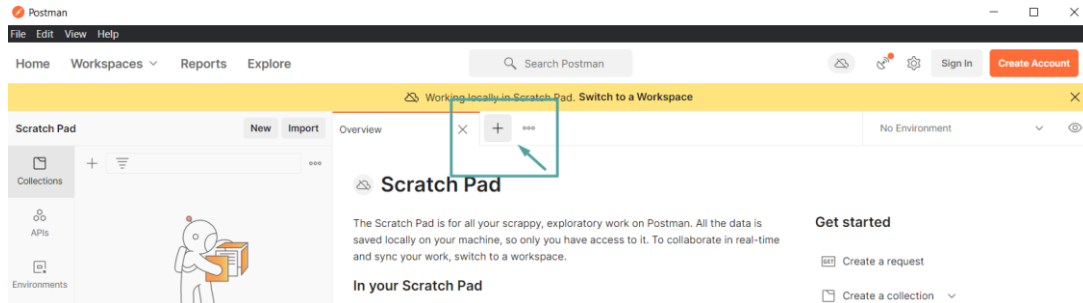
Y comenzamos con la instalación, haciendo doble clic en el archivo descargado.

Nombre	Fecha de modificación	Tipo	Tamaño
 Postman-win64-8.11.1-Setup.exe	27-08-2021 0:50	Aplicación	125.231 KB

Postman te pedirá crear una cuenta, pero puedes omitir este paso si lo deseas. Al crearla, permite guardar sesiones y otro tipo de características que pueden ser útiles para ti en un futuro, pero por ahora no será necesario.

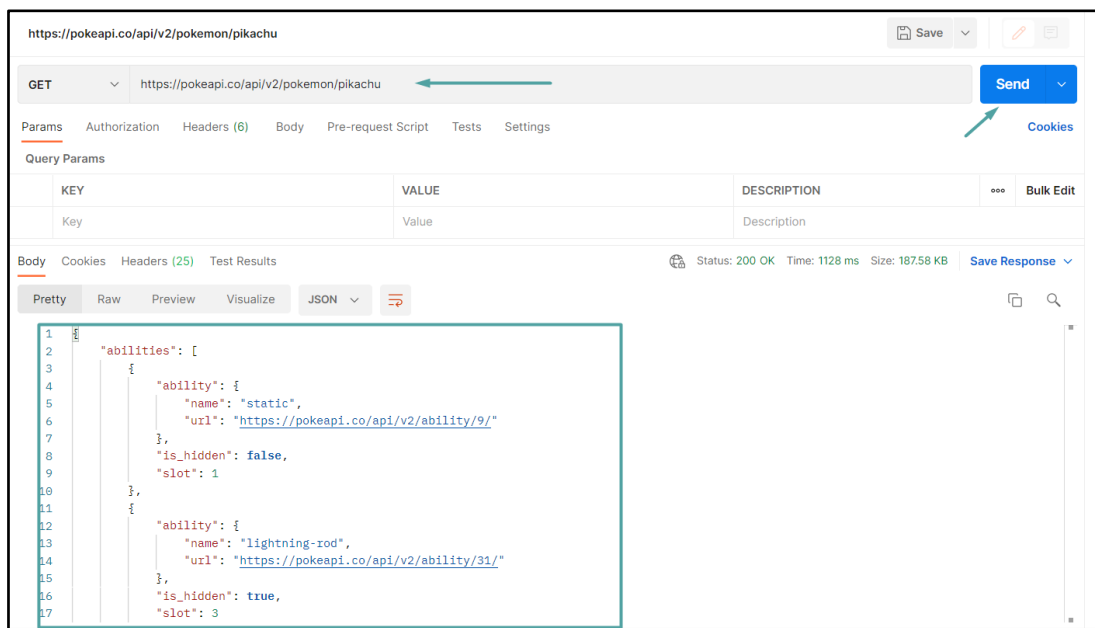


Una vez dentro de la aplicación, haremos clic en el signo más, para crear una nueva petición.



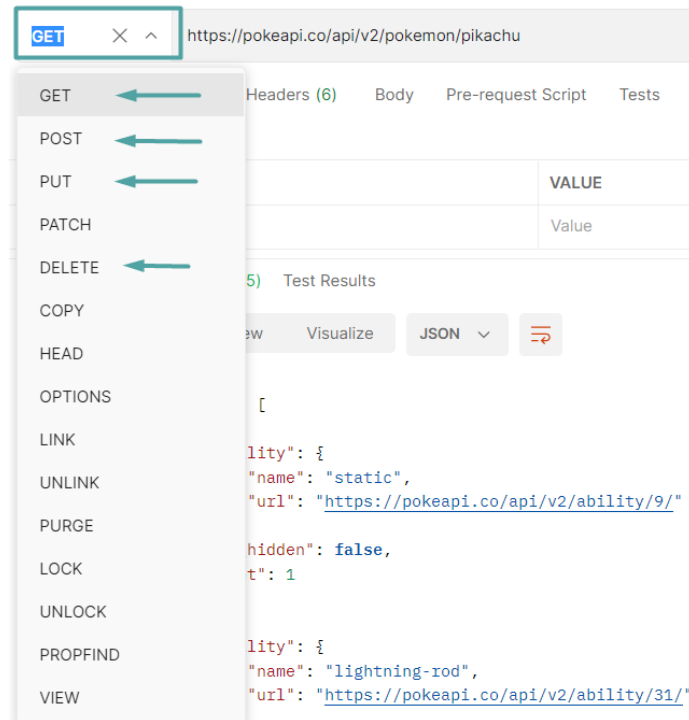
Y dentro del cuadro de texto, escribiremos la dirección en la que queremos realizar una petición HTTP. Utilizando la API de Pokémon, podemos ver la respuesta al consultar la URL:

<https://pokeapi.co/api/v2/pokemon/pikachu>



Otro detalle para notar es que al lado del cuadro de texto para ingresar la dirección URL, hay un menú deslizable, del cual podemos elegir los verbos HTTP que queremos utilizar para la petición en cuestión.

Como puedes ver, existen mucho otros verbos, pero solo nos enfocaremos en los cuatro básicos, de los que ya hemos hablado.



HACIENDO CONSULTAS A NUESTRO SERVIDOR LOCAL

Con **Postman**, podemos hacer consultas a nuestro servidor, lo cual nos libera de tener que usar el navegador para cada consulta, e incluso nos da la posibilidad de usar otros verbos **HTTP**. Utilizando el mismo código inicial de ejemplo, ejecutaremos el programa para levantar nuestro servidor.

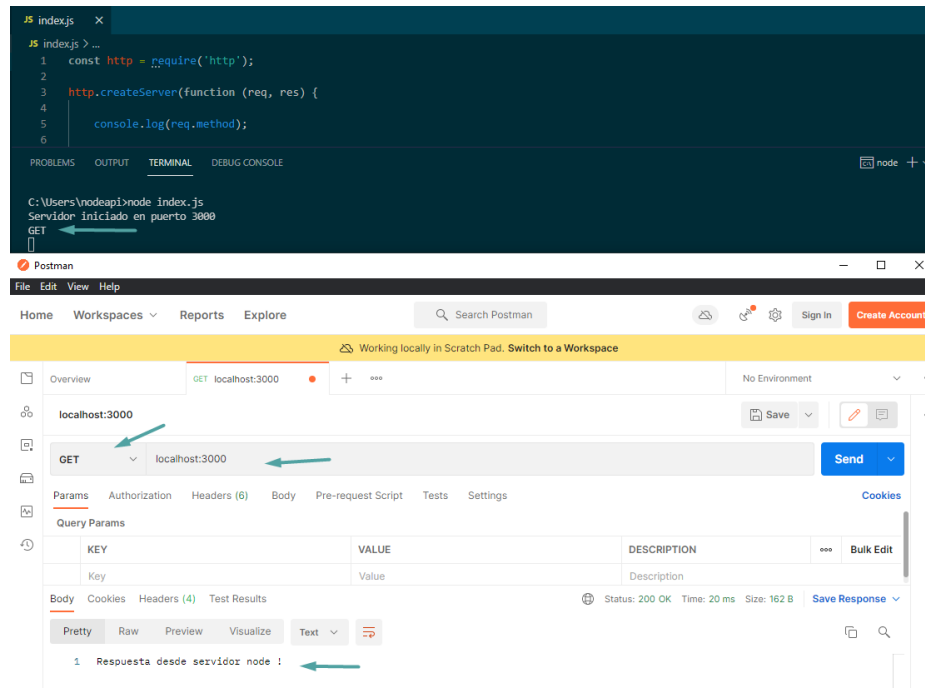
```

JS index.js  X
JS index.js > ...
1  const http = require('http');
2
3  http.createServer(function (req, res) {
4
5      console.log(req.method);
6
7      res.write("Respuesta desde servidor node !");
8      res.end();
9  })
10 .listen(3000, function(){
11     console.log("Servidor iniciado en puerto 3000");
12 });
13
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

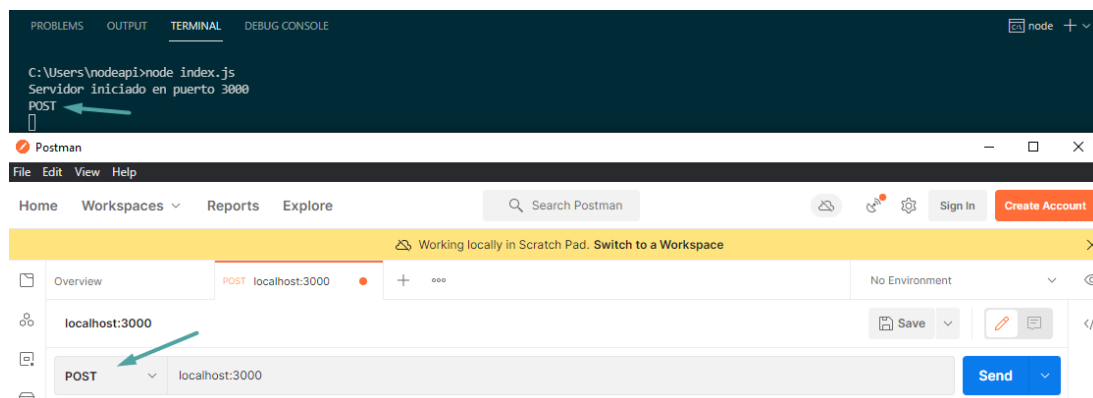
C:\Users\nodeapi>node index.js
Servidor iniciado en puerto 3000

```

Ahora que nuestro servidor está listo, realizaremos una consulta a la dirección localhost:3000.



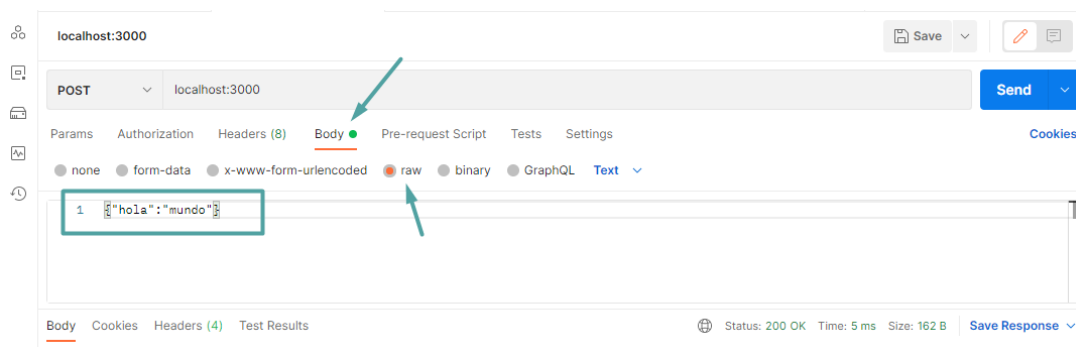
Cambiaremos el método **HTTP** desde **Postman**, y al enviar la petición, veremos que en nuestra consola aparecerá el método **HTTP** con el cual se está realizando la petición, lo que nos permitirá saber qué tipo de acción debe tomar nuestro programa.



PASANDO INFORMACIÓN DESDE LA PETICIÓN AL SERVIDOR

Cada vez que queremos hacer una petición que involucra envío de información, debemos encontrar la forma de procesarlo mediante la petición **HTTP**. Esto se puede lograr de varias maneras, una de ellas, es con el cuerpo o “body” de la petición.

Usando **postman**, podemos crear datos en formato **json**, en la pestaña **body** opción raw.



Luego, para recibir esta información, debemos utilizar el método **on()** sobre el objeto **request**, y así poder “escuchar” el momento en que llegan los datos al servidor. Con este método, podremos recibir los datos solo cuando estén llegando.

```
http.createServer(function (req, res) {  
  req.on('data', (body) => {  
    console.log(JSON.parse(body))  
  })  
})
```

A green arrow points to the `console.log(JSON.parse(body))` line in the code snippet.

Esto nos permite hacer la petición desde **postman**, y recibir el siguiente resultado.

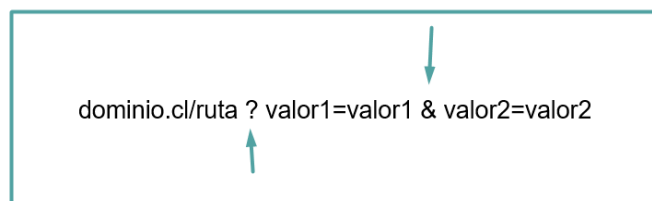
```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE  
  
C:\Users\nodeapi>node index.js  
Servidor iniciado en puerto 3000  
{ hola: 'mundo' }
```

A green arrow points to the JSON output `{ hola: 'mundo' }` in the terminal.

Otro detalle importante a mencionar, es que los datos recibidos desde una petición, llegan en grupos, por lo tanto, siempre es conveniente utilizar el método `on()` con el argumento `"end"`, el cual nos indica el momento en que la petición ha dejado de enviar datos al servidor. Solo allí será seguro utilizar los datos dentro de nuestro programa, ya que nos aseguramos de que la petición ha terminado, y no hay más datos por recibir. Con este código, aparentemente el resultado es el mismo, solo que esta vez estamos seguros de que no hemos perdido ningún dato en el camino.

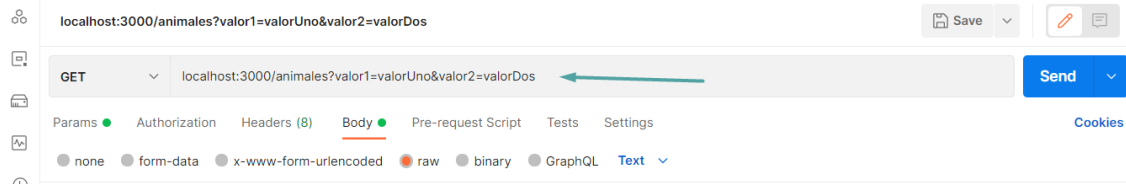
```
http.createServer(function (req, res) {
  let data;
  req.on('data', (body) => {
    data = JSON.parse(body);
  });
  req.on('end', () => {
    console.log(data);
  });
});
```

Como norma general, cuando queremos actualizar, eliminar, o consultar un objeto en específico, necesitaremos su id, o algún identificador único, que nos permita realizar acciones sobre él en particular, y ningún otro. Enviaremos este id por medio de la URL, utilizando **query strings**. **Query string** es un formato de cadena de texto para enviar consultas mediante la URL de una petición **HTTP**, y tiene la siguiente estructura:



Luego de la ruta de acceso a nuestro servidor, se abre la consulta o **query string** con el signo de pregunta `"?"`. Se agrega un juego de llave - valor, y en caso de querer agregar más valores, se deben separar con el símbolo ampersand `"&"`.

Para acceder a estos valores en nuestro servidor, primero debemos crear un nuevo objeto URL, al cual le pasaremos como parámetros, los datos obtenidos dentro del objeto **request**. Al hacer una petición desde **postman** con la siguiente URL: `localhost:3000/animales?valor1=valorUno&valor2=valorDos`



Y utilizando el siguiente código en nuestro servidor, obtenemos el siguiente resultado.

```

JS index.js x
JS index.js > http.createServer() callback
1  const http = require('http');
2
3  http.createServer(function (req, res) {
4    console.log(req.url);
5    console.log(req.headers.host);
6    const url = new URL(req.url, `http://${req.headers.host}`);
7    console.log(url);
8  }
9
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\nodeapi>node index.js
Servidor iniciado en puerto 3000
/animales?valor1=valorUno&valor2=valorDos
localhost:3000
URL {
  href: 'http://localhost:3000/animales?valor1=valorUno&valor2=valorDos',
  origin: 'http://localhost:3000',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'localhost:3000',
  hostname: 'localhost',
  port: '3000',
  pathname: '/animales',
  search: '?valor1=valorUno&valor2=valorDos',
  searchParams: URLSearchParams { 'valor1' => 'valorUno', 'valor2' => 'valorDos' },
  hash: ''
}
  
```

El objeto URL trae consigo información de gran utilidad para nosotros. Por ahora, nos centraremos en la propiedad `searchParams`. Ésta contiene otro objeto, llamado `URLSearchParams`, y aunque contiene los parámetros obtenidos de la `query string`, no podemos acceder directamente a ellos, pues primero debemos crear un nuevo objeto `URLSearchParams`, y pasarle como argumento la propiedad `searchParams` del objeto URL. Una vez creado este objeto, podemos acceder a los parámetros utilizando el método `get()`.

```

JS index.js x
JS index.js > http.createServer() callback
1  const http = require('http');
2
3  http.createServer(function (req, res) {
4    const url = new URL(req.url, `http://${req.headers.host}`);
5    const params = new URLSearchParams(url.searchParams);
6    console.log(params.get('valor1'));
7    console.log(params.get('valor2'));
8  });
9
10
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\nodeapi>node index.js
Servidor iniciado en puerto 3000
valorUno
valorDos
  
```

Consideremos el siguiente ejemplo, donde tenemos un objeto con las siguientes características, y que presenta dos animales distintos, cada uno con su id en particular.

```

1  const animales = {
2    1: {
3      animal: 'perro',
4      raza: 'siberian husky',
5      edad: 15
6    },
7    2: {
8      animal: 'gato',
9      raza: 'mainecoon',
10     edad: 20
11   }
12 }
  
```

Ahora, si quisiéramos solo leer los datos del objeto, podemos hacer la siguiente llamada **HTTP**, donde verificaremos primero que la ruta de entrada sea la correcta, y que el método sea igual a **GET**.

```

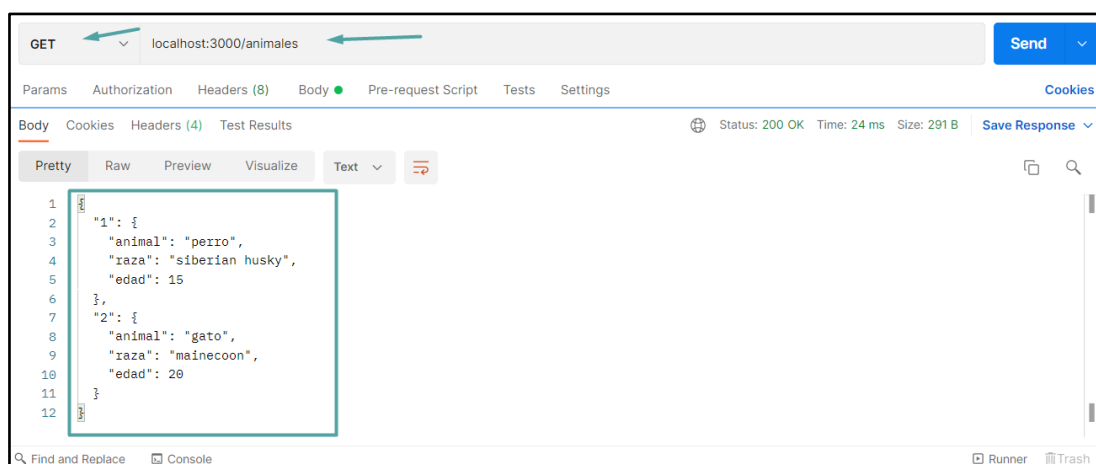
JS index.js x
JS index.js > ...
1  const http = require('http');
2  const animales = {
3    1: {
4      animal: 'perro',
5      raza: 'siberian husky',
6      edad: 15
7    },
8    2: {
9      animal: 'gato',
10     raza: 'mainecoon',
11     edad: 20
12   }
13 }
14 http.createServer(function (req, res) {
15   const url = new URL(req.url, `http://${req.headers.host}`);
16   const params = new URLSearchParams(url.searchParams);
17
18   if(req.url.startsWith('/animales') && req.method == 'GET'){
19     res.write(JSON.stringify(animales, null, 2));
20     res.end();
21   }
22 })
23
24 .listen(3000, function(){
25   console.log("Servidor iniciado en puerto 3000");
26 });
27

```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

C:\Users\nodeapi>

Utilizando **postman**, recibimos la siguiente respuesta.



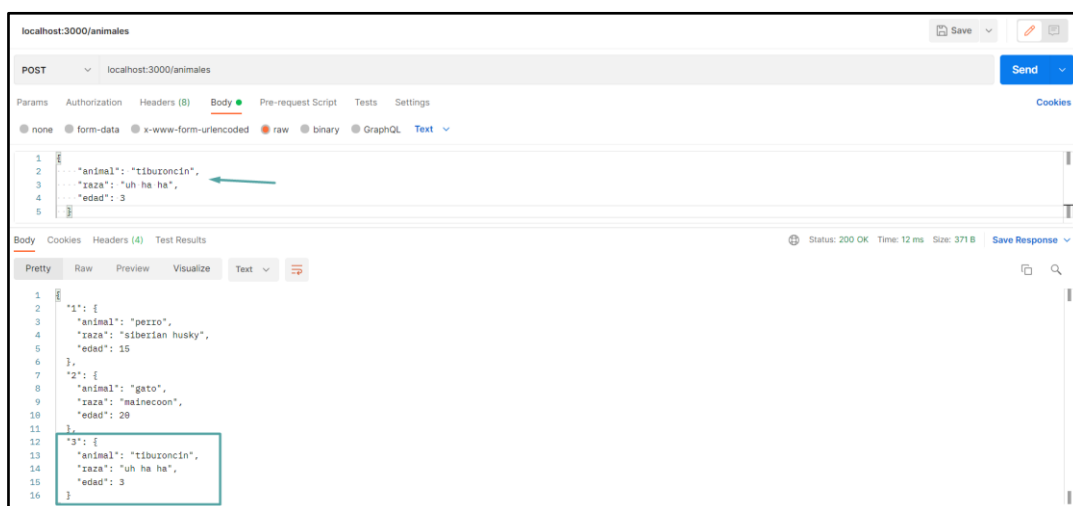
Si queremos agregar un nuevo animal, podemos utilizar el siguiente código.

```
http.createServer(function (req, res) {
  const url = new URL(req.url, `http://${req.headers.host}`);
  const params = new URLSearchParams(url.searchParams);

  if(req.url.startsWith('/animales') && req.method == 'POST'){
    let nuevoAnimal;
    req.on('data', (datos) => {
      nuevoAnimal = JSON.parse(datos);
    })
    req.on('end', ()=> {
      const nuevoId = Object.keys(animales).length + 1 //Obtener numero de objetos y sumar 1
      animales[nuevoId] = nuevoAnimal
      res.write(JSON.stringify(animales,null,2))
      res.end();
    })
  })
})

.listen(3000, function(){
  console.log("Servidor iniciado en puerto 3000");
});
```

Lo cual, desde **postman**, nos da la siguiente respuesta.



The screenshot shows a Postman interface with a POST request to `localhost:3000/animales`. The request body is a JSON object:

```
{
  "animal": "tiburoncin",
  "raza": "uh ha ha",
  "edad": 3
}
```

The response body is a JSON array of three animal objects:

```
[
  {
    "1": {
      "animal": "perro",
      "raza": "siberian husky",
      "edad": 15
    },
    "2": {
      "animal": "gato",
      "raza": "mainecoon",
      "edad": 20
    },
    "3": {
      "animal": "tiburoncin",
      "raza": "uh ha ha",
      "edad": 3
    }
  ]
}
```

Si quisiéramos modificar, o actualizar algún objeto, podemos utilizar la siguiente sintaxis en el servidor, donde rescataremos el id que pasaremos por la **query string** en la URL.

```
http.createServer(function (req, res) {
  const url = new URL(req.url, `http://${req.headers.host}`);
  const params = new URLSearchParams(url.searchParams);

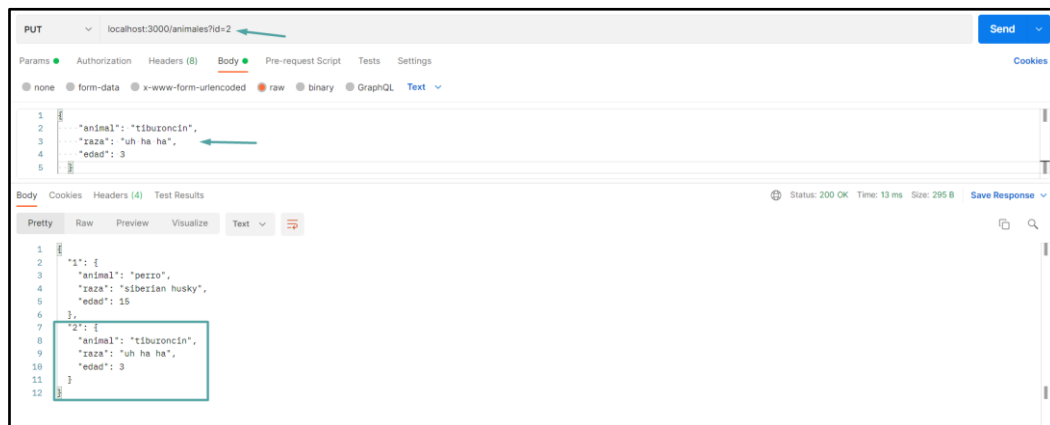
  if(req.url.startsWith('/animales') && req.method == 'PUT'){
    let animal;
    req.on('data', (datos) => {
      animal = JSON.parse(datos);
    })
    req.on('end', () => {
      const id = params.get('id');

      const animalActualizado = { ...animales[id], ...animal } //Actualizando animal

      animales[id] = animalActualizado; //Redefiniendo animal dentro de objeto inicial animales

      res.write(JSON.stringify(animales,null,2))
      res.end();
    })
  }
})
.listen(3000, function(){
  console.log("Servidor iniciado en puerto 3000");
});
```

Y luego, haciendo la siguiente petición HTTP desde **Postman**, obtendremos el siguiente resultado.



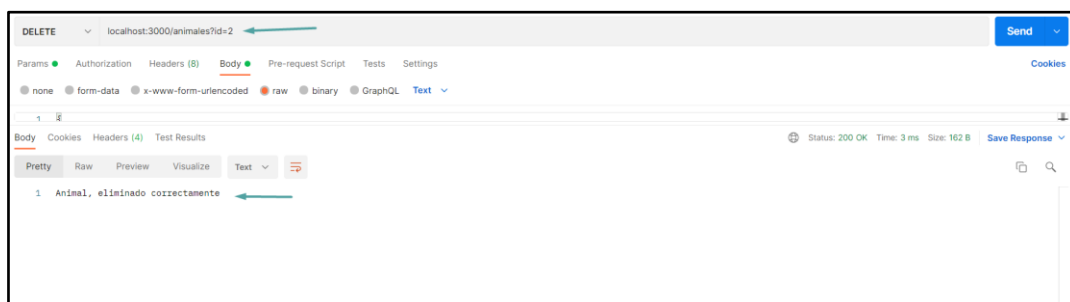
De manera similar, podemos eliminar un objeto, pasando su id por la URL de la petición. En general, cuando esto se realiza, solo se envía una confirmación de vuelta al usuario. Podemos utilizar este código para el servidor.

```

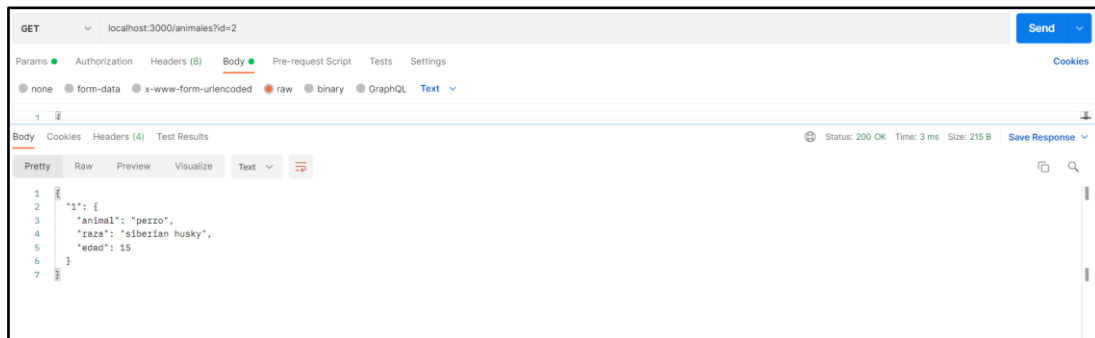
JS index.js x
JS index.js > http.createServer() callback
14 http.createServer(function (req, res) {
15     const url = new URL(req.url, `http://${req.headers.host}`);
16     const params = new URLSearchParams(url.searchParams);
17
18     if(req.url.startsWith('/animales') && req.method == 'GET'){
19         res.write(JSON.stringify(animales, null, 2));
20         res.end()
21     }
22
23     if(req.url.startsWith('/animales') && req.method == 'DELETE'){
24         const id = params.get('id');
25
26         delete animales[id];
27
28         res.write("Animal, eliminado correctamente");
29         res.end();
30     }
31 }
32 })
33 .listen(3000, function(){
34     console.log("Servidor iniciado en puerto 3000");
35 });
36

```

Y realizar la petición.



Si volvemos a consultar por todos los animales, podemos ver que el animal con id 2 ha sido eliminado correctamente.



Como ya hemos mencionado antes, estas operaciones están siendo realizadas sobre un objeto ya definido, por lo tanto, cuando reiniciemos nuestro servidor, todos los cambios realizados a este objeto se perderán. En el próximo ejercicio, utilizaremos los mismos métodos para guardar datos dentro de archivos de texto, pero esta vez utilizando nuestro servidor.