



**Argentina
programa
4.0**



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

***primero
la gente***

Clase 9: Api Restful y Json

Agenda de hoy

- A. Qué es API Restful
- B. Cómo simular un API Restful
 - a. El ejemplo de JSON Placeholder
- C. Peticiones
 - a. GET genérico
 - b. GET específica
 - c. POST
 - d. PUT
 - e. DELETE
 - f. PATCH
- D. Limitaciones en peticiones
 - Los nuevos actores
- A. Proyecto integrador: pautas del primer proyecto



API Restful y JSON

El trayecto que hemos transitado hasta aquí, es muy breve en el mundo de la programación backend pero, aún así, ya hemos trabajado con un sinfín de herramientas y tecnologías.

Las prácticas abordadas hasta el momento fueron pensadas de cara a que puedas asimilar paulatinamente las técnicas claves de aplicaciones Node.js, para poder abordar tu primer proyecto integrador.



API Restful y JSON

En este nuevo encuentro, salimos de una API convencional que solo nutre de información a aquellas aplicaciones frontend que solicitan la misma, para conocer el poder completo del término API Restful.

Por ello, hoy abordaremos las principales herramientas que nos provee el framework Express JS, para poder administrar una aplicación de backend, convirtiendo a ésta en una aplicación 100% funcional.



API Restful y JSON

Si bien nos falta un componente clave aún en el mundo backend, que son las bases de datos, aprovecharemos características del Node.js como ser FileSystem API junto al formato de transporte de datos que nos ofrece JSON, para simular esa base de datos que aún no tenemos, utilizando un archivo JSON para poder no solo peticionar productos, sino también para agregar nuevos productos.



API Restful y JSON

Además de esto último, conoceremos de qué forma se utiliza API Restful para modificar productos existentes y también para eliminar productos que tal vez ya no son necesarios.

Veamos entonces de qué trata esto, y cuáles son las herramientas de Express JS para plasmar este conocimiento en una aplicación backend real.



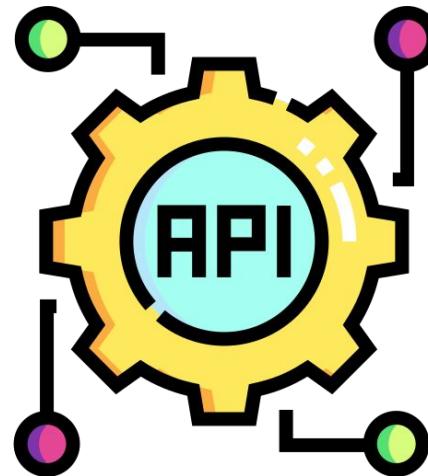
Qué es API Restful

Qué es API Restful

Si bien lo hemos hablado en encuentros anteriores, vamos a repasar desde cero los fundamentos de API Restful.

Una API Restful es un conjunto de principios de diseño para servicios web que se utilizan para crear servicios web ligeros, escalables y mantenibles.

El término "*RESTful*" se refiere al término "*Transferencia de Estado Representacional*" **Representational State Transfer** y basado en el protocolo HTTP (**Hypertext Transfer Protocol**).



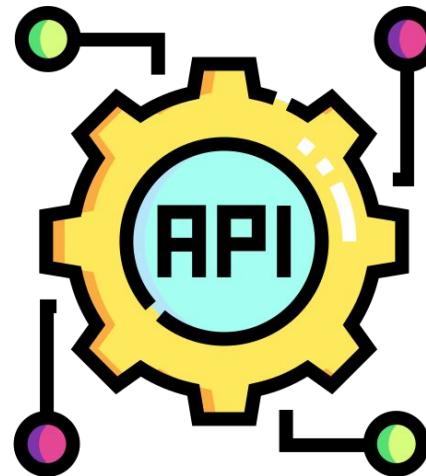
Qué es API Restful

Las APIs Restful permiten a los clientes interactuar con los servidores web mediante solicitudes HTTP estándar, como:

- GET
- POST
- PUT
- DELETE

etc.

Estas solicitudes HTTP se envían a una URL específica (**endpoint**), y el servidor web responde con un recurso que representa el estado actual del recurso solicitado.



Qué es API Restful

Los principios fundamentales de una API Restful son:

Principios

Utilizar URIs (*Uniform Resource Identifiers*) para identificar recursos. Cada recurso debe tener una identificación única que se exprese a través de una URI.

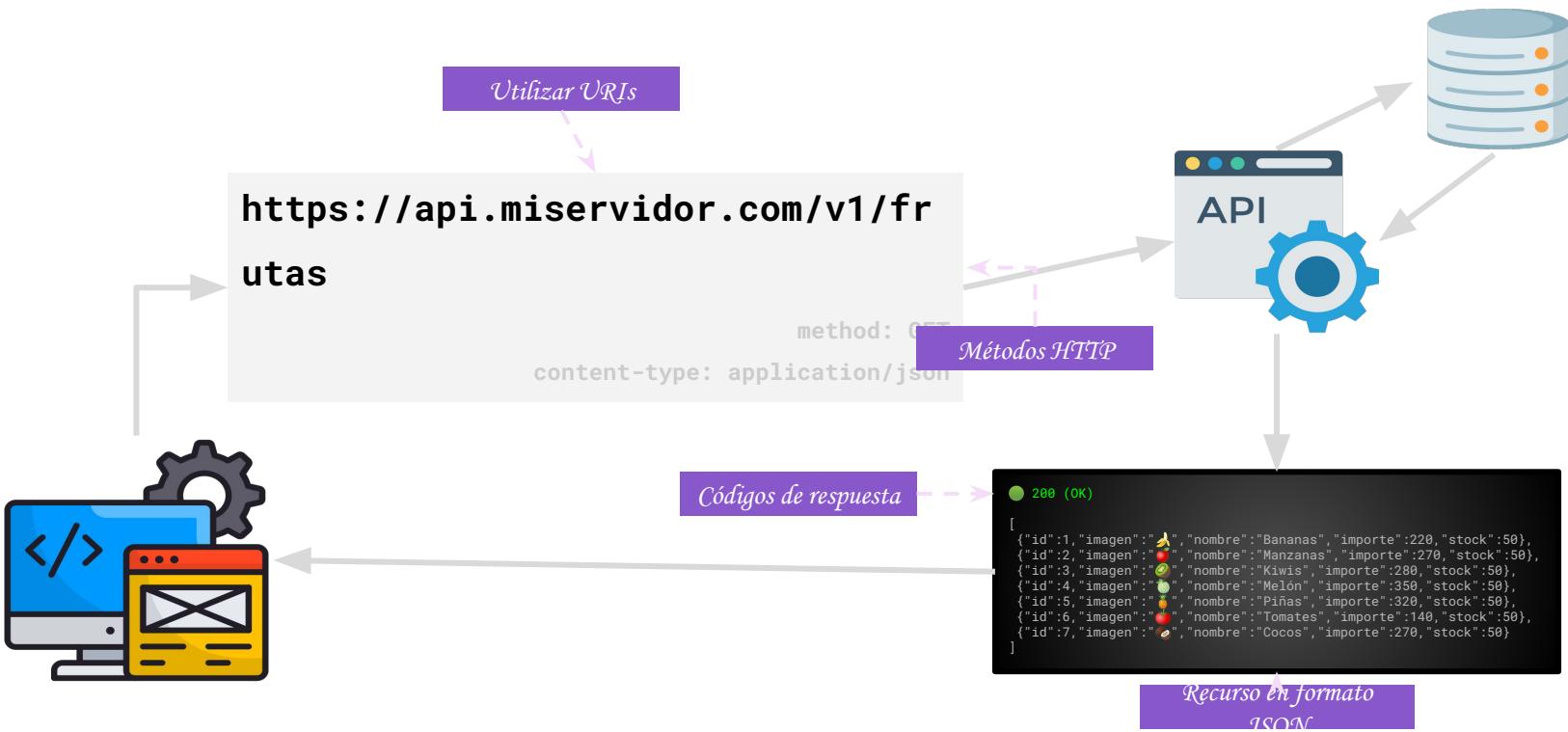
Utilizar métodos HTTP estándar (GET, POST, PUT, DELETE, *entre otros*) para operar sobre los recursos.

Utilizar los códigos de respuesta HTTP para indicar el estado de la solicitud.

Ser independiente del lenguaje de programación utilizado en el servidor y en el cliente.

Permitir la **representación de los recursos en formatos** diferentes, como **JSON o XML**.

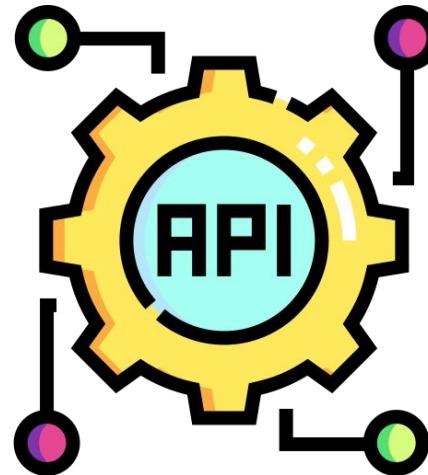
Qué es API Restful



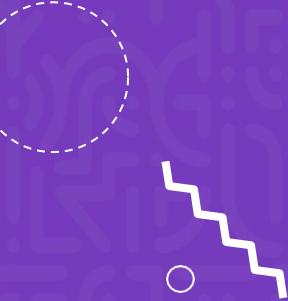
Qué es API Restful

En toda esta ecuación, nuestra aplicación BACKEND es quien tiene el rol principal de:

- garantizar el acceso a los datos
- dialogar con la bb.dd. o identificar el recurso que llegó en la petición
- adaptar el mismo para ser transferido bajo una respuesta
- informar el estado de la petición efectivo o erróneo, según la disponibilidad o no de dichos recursos



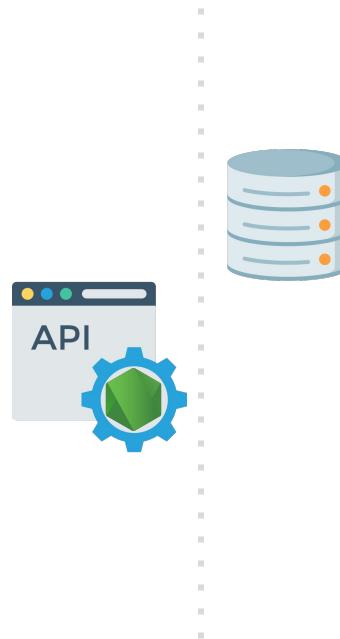
Como simular una API Restful



Cómo simular un API Restful

Con los conocimientos y experiencia adquiridos, estamos en condiciones de desarrollar nuestra primera API Restful.

Tendremos limitaciones en su construcción pero, la idea principal, es cerrar el concepto de la experiencia no solo de servir datos a clientes, sino también de poder grabar datos desde un cliente en nuestra aplicación de servidor.



Una de las limitaciones que tenemos hasta el momento, es integrar una base de datos real.

Más allá de esto, junto con las características de JavaScript aprendidas (*FileSystem, JSON format, Express JS*), **simularemos la base de datos en un archivo JSON** a la cual le pediremos datos y también los almacenaremos.

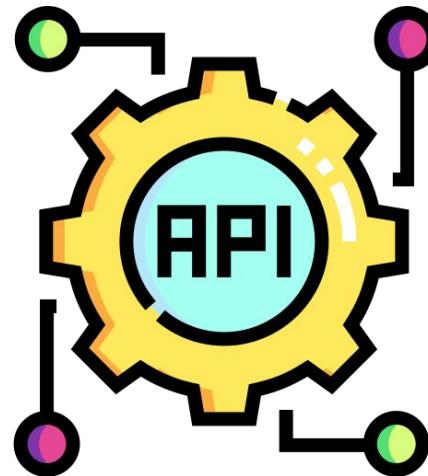


Cómo simular un API Restful

Trabajaremos con tres peticiones básicas utilizando API Restful para poder visualizar la información del archivo en formato JSON.

- GET
- GET:id
- POST

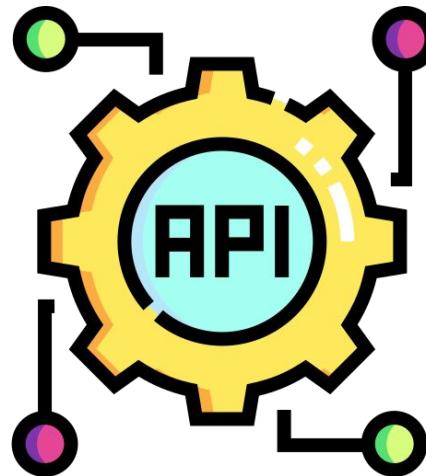
La primera de ellas, **GET**, nos devolverá toda la información de los productos en cuestión.



Cómo simular un API Restful

La segunda petición, **GET:id**, nos permitirá obtener, en este caso, un solo producto buscando el mismo por su código, o Identificador.

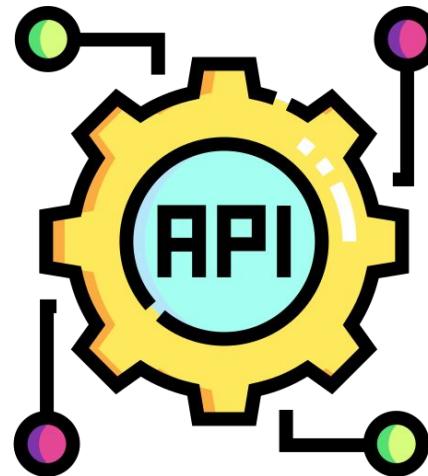
En este escenario obtendremos solo un producto, o ninguno, de acuerdo a su ID. No aplicaremos un filtro como hicimos anteriormente, dado que buscaremos por un valor numérico exacto.



Cómo simular un API Restful

La tercera opción, **POST**, la utilizaremos para grabar nuevos productos a través de la **herramienta de API Testing Thunder Client**.

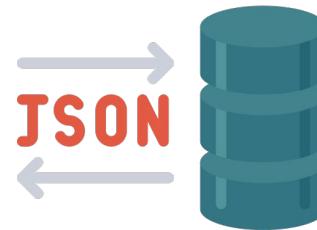
En esta opción, entrarán en escena nuevos actores que nos ayudarán a controlar los diferentes aspectos del envío y recepción de datos. La extensión **Thunder Client**, será nuestro cliente, o sea, el reemplazo de una aplicación frontend, que nos ayudará a enviar datos y recibir respuestas.



El ejemplo de JSON Placeholder

El ejemplo de JSON Placeholder

Con este último método HTTP (**POST**) entrarán en escena nuevos actores quienes nos ayudarán a controlar diferentes aspectos necesarios, para el envío y recepción de datos. La extensión [**Thunder Client**](#), será nuestro cliente, (*el reemplazo de una aplicación frontend*), que nos ayudará a enviar datos y recibir respuestas.



El ejemplo de JSON Placeholder

Desde la sección [Guide de la web oficial de JSON](#)

[Placeholder](#), nos concentraremos específicamente en las rutas de API Restful informadas al final de dicha página.

Aquí contamos con varias APIs con datos ficticios que muestran textos en [formato LOREM IPSUM](#), más algunas otras propiedades con datos complementarios.

Esto es suficiente para entender el funcionamiento de los diferentes métodos HTTP que podemos utilizar.

The available nested routes are:

- [/posts/1/comments](#)
- [/albums/1/photos](#)
- [/users/1/albums](#)
- [/users/1/todos](#)
- [/users/1/posts](#)

[You can sponsor this project \(and others\) on GitHub](#)

Coded and maintained with ❤ by [typicode](#) © 2022



El ejemplo de JSON Placeholder

Aquí podemos apreciar la estructura o formato con el cual nos responderá el endpoint POSTS de JSON Placeholder:

<https://jsonplaceholder.typicode.com/posts>

Veamos cómo se comporta esta petición realizando la misma desde **Thunder Client**. Si bien ya trabajamos con peticiones **GET** en nuestros encuentros anteriores, siempre conviene refrescar conocimientos porque cada Endpoint puede tener sus particularidades en responder con datos a nuestra solicitud.

```
JSON Placeholder >> POSTS
[  
  {  
    "userId": 1,  
    "id": 1,  
    "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",  
    "body": "quia et suscipit\\nsuscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut ut quas totam\\nnostrum rerum est autem sunt rem eveniet architecto"  
  },  
  {  
    "userId": 1,  
    "id": 2,  
    "title": "qui est esse",  
    "body": "est rerum tempore vitae\\nsequi sint nihil reprehenderit dolor beatae ea dolores neque\\nrefugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\\nqui aperiam non debitis possimus qui neque nisi nulla"  
  }...  
]
```

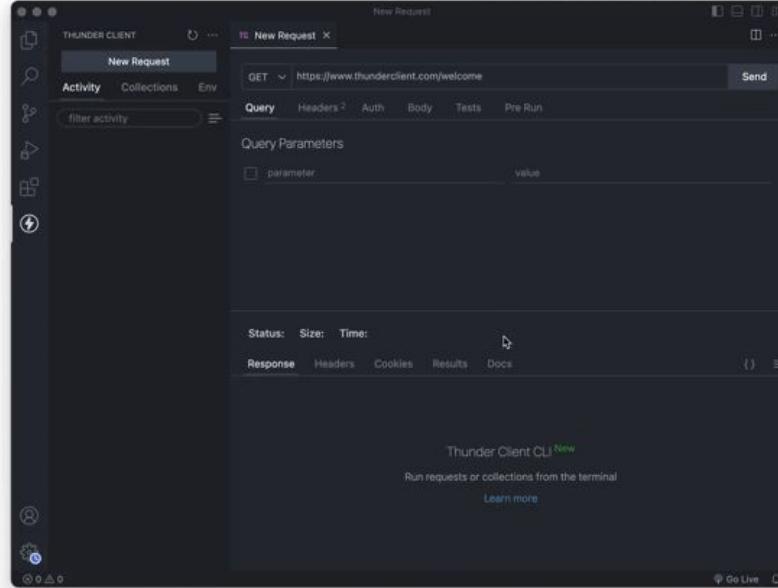


Petición GET

Petición GET

Al ejecutar la **petición GET** en el endpoint **JSON Placeholder /POSTS**, este nos retorna un set de datos, tal como hemos visto anteriormente.

Además de dicho set de datos, el servidor siempre retorna información adicional la cual podemos ver en el panel **headers**.



Petición GET específica

Petición GET específica

En el caso que deseemos retornar solo un dato de todo el conjunto, utilizamos la petición específica, la cual recibe por parámetro el valor que necesitamos para filtrar.

En este caso, el parámetro predeterminado es la propiedad ID, y la informamos en la URL agregando */:n* para así obtener el resultado deseado.

Header	Value
Date	Fri, 31 Mar 2023 15:00:56 GMT
Content-Type	application/json; charset=utf-8
Transfer-Encoding	chunked
Connection	close
X-Powered-By	Express
X-RateLimit-Limit	1000
X-RateLimit-Remaining	999
X-RateLimit-Reset	1658970447
Vary	Origin, Accept-Encoding
Access-Control-Allow-Credentials	true
Cache-Control	max-age=43200
Pragma	no-cache



Petición GET específica

The screenshot shows the Postman application interface. At the top, it says "tc New Request X". Below that, there's a dropdown set to "GET" and a URL input field containing "https://jsonplaceholder.typicode.com/posts/7", which is highlighted with a pink rectangle. To the right of the URL is a "Send" button. Below the URL, there are tabs for "Query", "Headers 2", "Auth", "Body", "Tests", and "Pre Run", with "Query" being the active tab. Underneath these tabs, the status of the request is shown as "Status: 200 OK Size: 225 Bytes Time: 507 ms". At the bottom, there are tabs for "Response", "Headers 23", "Cookies", "Results", and "Docs", with "Response" being the active tab. There are also some icons at the bottom right.

Seleccionamos un ID de todo el listado de POSTS y lo indicamos en la URI utilizando lo que aprendimos y trabajamos como URL Params.

Petición POST

Petición POST

El método **POST** se utiliza para crear nuevos recursos en el servidor. El mismo requiere no solamente definir como POST el método a enviar, sino que también utilicemos el cuerpo (**body**) de la petición para que enviamos en éste, la información del recurso a crear.

JSON Placeholder nos permite simular el agregado de un recurso, tal como vemos en la imagen contigua.

The screenshot shows the Thunder Client interface with a successful POST request to `https://jsonplaceholder.typicode.com/posts`. The response status is `200 OK`, size is `26.88 KB`, and time is `115 ms`. The Headers tab displays the following response headers:

Header	Value
date	Fri, 31 Mar 2023 15:00:56 GMT
content-type	application/json; charset=utf-8
transfer-encoding	chunked
connection	close
x-powered-by	Express
x-ratelimit-limit	1000
x-ratelimit-remaining	999
x-ratelimit-reset	1658970447
vary	Origin, Accept-Encoding
access-control-allow-credentials	true
cache-control	max-age=43200
pragma	no-cache



Petición POST

The screenshot shows the Postman application interface. At the top, it says "TC New Request X". Below that, the method is set to "POST" and the URL is "https://jsonplaceholder.typicode.com/posts". There are tabs for "Query", "Headers 2", "Auth", "Body 1", "Tests", and "Pre Run". The "Body" tab is selected, showing "Json" as the format. Under "Json Content", there is a code block:

```
1 {  
2   "userId": 1,  
3   "title": "NUEVO POST DE PRUEBA",  
4   "body": "Este es un post de prueba para grabar en JSON Placeholder. El mismo no se graba de  
verdad. JSON Placeholder simula su grabación, y nos retornará como resultado el mismo POST  
enviado, con el agregado de la propiedad ID, y el número de identificador que le  
correspondería."  
5 }
```

Seleccionamos el método POST, y la URI base de JSON Placeholder.

En la sección Body, agregamos la estructura del recurso a crear: JSON en nuestro caso, respetando el formato del dato.

Cuando los servidores deben resolver el ID que le corresponde a cada nuevo POST, podemos obviar esta información, aunque sí debemos tener en cuenta que la lógica de nuestra aplicación backend debe estar preparada para resolver este punto.



Petición POST

TC New Request ×

Status: 201 Created Size: 353 Bytes Time: 474 ms

Response Headers 24 Cookies Results Docs { } ≡

```
1  [
2   "userId": 1,
3   "title": "NUEVO POST DE PRUEBA",
4   "body": "Este es un post de prueba para grabar en JSON Placeholder. El mismo no se graba de verdad.
      JSON Placeholder simula su grabación, y nos retornará como resultado el mismo POST enviado, con
      el agregado de la propiedad ID, y el número de identificador que le correspondería.",
5   "id": 101
6 ]
```

Una vez que el servidor crea el recurso de acuerdo a los datos enviados en el cuerpo de la petición, suele retornar un mensaje.

Puede ser un mensaje de confirmación simple o, como en el caso de **JSON Placeholder**, nos retorna el mismo recurso creado con el agregado de la propiedad **ID** y el número que se le asignó.

En el **Código de Estado** encontraremos el número **201**, correspondiente a **Recurso Creado**.



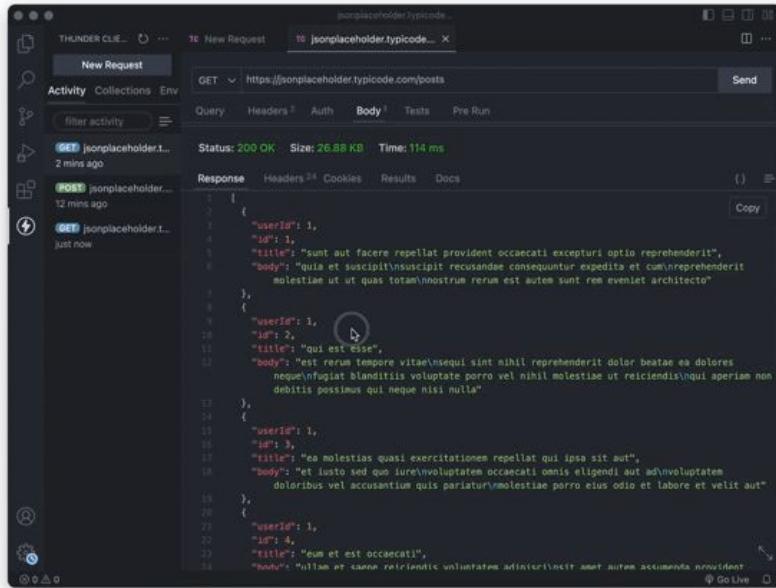
Petición PUT

Petición PUT

El método **PUT** se utiliza tanto para crear nuevos recursos en el servidor como también para modificar recursos existentes.

Requiere también que informemos en el cuerpo (**body**) de la petición el recurso a modificar.

En el caso de la API de JSON Placeholder, debemos informar también en la URI, el ID del recurso a modificar.



The screenshot shows the Thunder Client interface with a PUT request to the URL `https://jsonplaceholder.typicode.com/posts/1`. The response status is 200 OK, size is 26.88 KB, and time is 114 ms. The response body is a JSON array of posts, with the first post's properties highlighted:

```
[{"userId": 1, "id": 1, "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit", "body": "quia et suscipit\\n suscipit recusandae consequuntur expedita et cum\\n reprehenderit molestiae ut quas totam\\n nostrum rerum est autem sunt rem eveniet architecto"}, {"userId": 1, "id": 2, "title": "qui est esse", "body": "est rerum tempore vitae\\n sequi sint nihil reprehenderit dolor beatae ea dolores neque\\n fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\\n qui aperiam non debitis possimus qui neque nisi nulla"}, {"userId": 1, "id": 3, "title": "ea molestias quasi exercitationem repellat qui ipsa sit aut", "body": "et iusto sed quo iure\\n voluntatem occaecati omnis eligendi aut ad\\n voluptatem doloribus vel accusantium quis pariatur\\n molestiae porro eius odio et labore et velit aut"}, {"userId": 1, "id": 4, "title": "eum et est occaecati", "body": "William et ea ratione\\n retrivit\\n voluntatem administratively\\n smat\\n autem accumsan accident"}]
```



Petición POST

The screenshot shows a Postman request configuration. The method is set to **PUT**, the URL is <https://jsonplaceholder.typicode.com/posts/2>, and the **Body** tab is selected. Under the **Body** tab, the **JSON** tab is active, and the JSON Content is:

```
1  "userId": 1,
2  "title": "Modifico el título del POST",
3  "body": "est rerum tempore Vitae\nsequi sint nihil reprehenderit dolor beatae ea dolores
neque\\n fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\\nqui aperiam
non debitis possimus qui neque nisi nulla"
5 }
```

En el cuerpo de la petición estructuramos el recurso que enviaremos al servidor, con el cambio correspondiente en su propiedad **title**. También podemos notar que en la estructura del recurso obviamos informar el **ID** del mismo.

En la **URI** definimos el método **PUT**, y agregamos, de acuerdo a cómo lo solicita JSON Placeholder, el **ID del recurso** que modificaremos.



Petición POST

Status: 200 OK Size: 294 Bytes Time: 415 ms

Response Headers²³ Cookies Results Docs { } ⚙

```
1 {
2   "userId": 1,
3   "title": "Modifico el título del POST",
4   "body": "est rerum tempore vitae\\nsequi sint nihil reprehenderit dolor beatae ea dolores
      neque\\nfugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\\nqui aperiam non
      debitis possimus qui neque nisi nulla",
5   "id": 2
6 }
```

Cuando el servidor modifica el recurso de acuerdo a los datos enviados en el cuerpo de la petición, al igual que el método POST, nos retorna un mensaje notificando la tarea.

Puede ser también un mensaje de confirmación simple o, como en el caso de **JSON Placeholder**, nos retorna el mismo recurso creado agregando el ID correspondiente y reflejando el cambio solicitado en el resto de la estructura.

En el **Código de Estado** encontraremos el número **200**, correspondiente a **OK**.

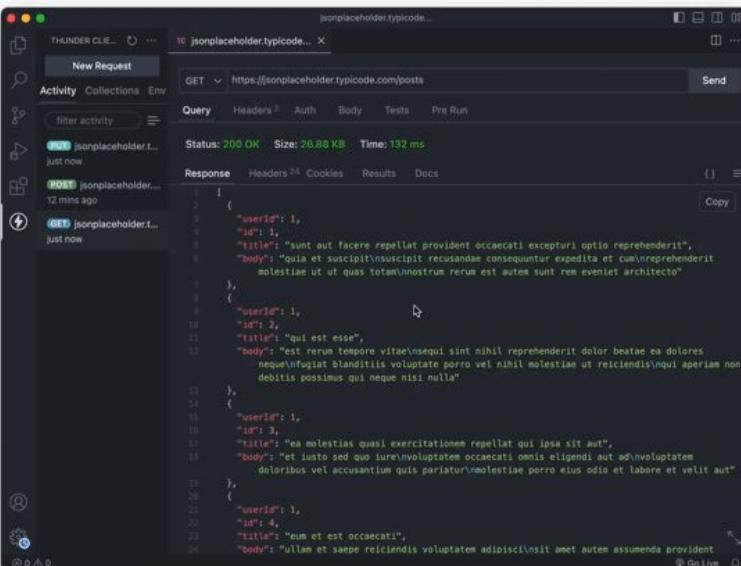


Petición DELETE

Petición DELETE

El método **DELETE**, tal como su nombre lo indica, se utiliza para eliminar recursos existentes en el servidor.

En este caso, no requiere que informemos nada en el cuerpo (**body**) de la petición. Solo debemos pasar por URL Params, el ID del post que deseamos eliminar.



The screenshot shows a POST request to `https://jsonplaceholder.typicode.com/posts` with an ID of 1. The response is a JSON array of four posts, with the first post matching the request parameters. The JSON output is as follows:

```
[{"id": 1, "userId": 1, "title": "wunt aut facere repellat provident occaecati excepturi optio reprehenderit", "body": "quia et suscipit\\nscipit recusandae consequuntur expedita et cum\\nreprehenderit molestiae ut quas totam\\nnostrum rerum est autem sunt rem eveniet architecto"}, {"id": 2, "userId": 1, "title": "qui est esse", "body": "est rerum tempore vitae\\nsequi sint nihil reprehenderit dolor beatae ea dolores neque\\nfugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis\\nqui aperiam non debitis possimus qui neque nisi nulla"}, {"id": 3, "userId": 1, "title": "ea molestias quasi exercitationem repellat qui ipsa sit aut", "body": "et iusto sed quo iure\\nvolutatem occaecati omnis eligendi aut ad\\nvolutatem doloribus vel accusantium quis pariatur\\nmolestiae porro eius odio et labore et velit aut"}, {"id": 4, "userId": 1, "title": "eum et est occaecati", "body": "ullam et saepe reiciendis voluptatem adipisci\\nsit amet autem assumenda provident\" data-bbox="456 248 842 772"/>
```



Petición DELETE

The screenshot shows the Postman application interface. At the top, there's a header bar with the title 'TC jsonplaceholder.typicode...'. Below it, the main interface has a 'DELETE' method selected and a URL 'https://jsonplaceholder.typicode.com/posts/13' entered. A 'Send' button is to the right. Underneath, there are tabs for 'Query', 'Headers 2', 'Auth', 'Body', 'Tests', and 'Pre Run'. The 'Response' tab is active, displaying the results of the request. It shows 'Status: 200 OK', 'Size: 2 Bytes', and 'Time: 508 ms'. The response body is shown as a single line: '1 {}'. The entire URL input field and the response area are highlighted with a pink rectangular border.

En la URI enviamos el **ID** del post a eliminar.

Al ejecutar el método **DELETE**, el servidor nos devuelve como resultado el código de respuesta **200 - OK**, y un objeto vacío.



Petición PATCH

Petición PATCH

The screenshot shows a POSTMAN interface with the following details:

- METHOD: PATCH
- URL: <https://jsonplaceholder.typicode.com/posts/13>
- Body tab is selected.
- JSON Content:

```
1  {
2    "title": "Modifico el título del POST no. 13"
3 }
```

El método PATCH es muy poco utilizado, igual es bueno tener presente cómo funciona: Está creado para modificar un recurso de servidor, pero con la particularidad de solo tener que indicar la parte del recurso a modificar en el cuerpo de la petición.

Luego, en la URL, informamos vía URL Params el identificador del recurso a modificar.



Petición PATCH

```
Status: 200 OK  Size: 321 Bytes  Time: 435 ms

Response Headers 23 Cookies Results Docs { } ≡

1 {
2   "userId": 2,
3   "id": 13,
4   "title": "Modifico el título del POST no. 13",
5   "body": "aut dicta possimus sint mollitia voluptas commodi quo doloremque\niste corrupti reiciendis
    voluptatem eius rerum\nsit cumque quod eligendi laborum minima\nperferendis recusandae assumenda
    consectetur porro architecto ipsum ipsam"
6 }
```

Como resultado, el servidor informa mediante el código de estado **200 - OK**, que el recurso fue modificado correctamente. En algunos casos, como por ejemplo en el servidor de JSON Placeholder, el servidor nos devuelve el recurso completo con la modificación aplicada.

Alternativamente, otros servidores pueden devolver un mensaje de confirmación junto al código de estado de la operación.



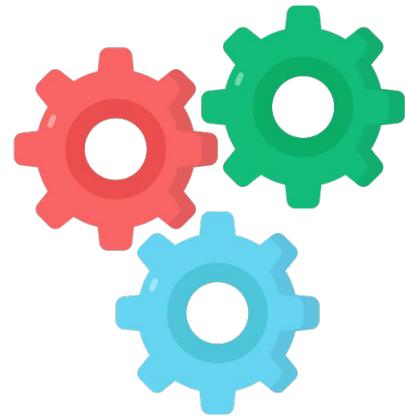
Limitaciones en peticiones

Los nuevos actores

Middleware

Se denomina **Middleware** a un software que actúa como intermediario entre diferentes aplicaciones o sistemas.

En el contexto de las aplicaciones web, el Middleware se refiere a un conjunto de funciones que se ejecutan entre la solicitud del cliente y la respuesta del servidor.

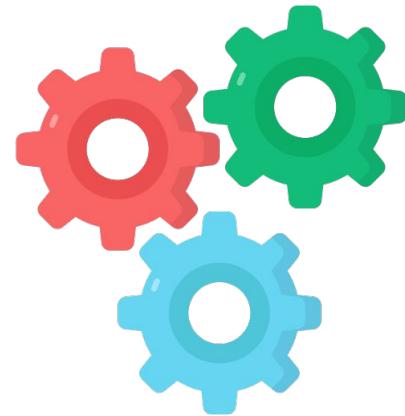


Middleware

Se utiliza comúnmente para realizar tareas como:

- autenticación del usuario
- análisis de datos de solicitud
- gestión de sesiones
- manejar la caché de contenido

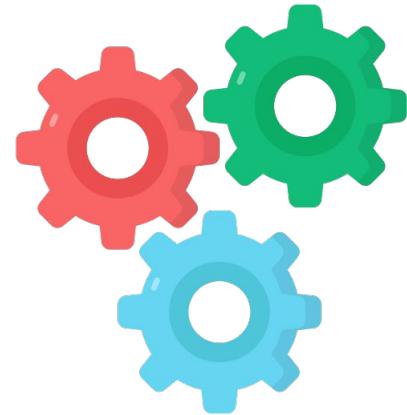
Cada función de Middleware se ejecuta en orden y puede modificar la solicitud o la respuesta según sea necesario.



Middleware

En nuestro proyecto de ejemplo, el Rol de Middleware será abordado por la herramienta **Body-Parser**, la cual hablaremos a continuación, y por el manejo del contenido JSON en el archivo **frutas.json** que oficialará como base de datos.

En ambas situaciones estamos sumando procesos independientes a la actividad principal de nuestro proyecto (escuchar peticiones HTTP), por lo tanto, es clave hacer esto utilizando Middlewares de acuerdo a la necesidad de cada caso.



Body Parser

body-parser es un middleware de Node.js que se utiliza para analizar los datos de la solicitud entrante en el servidor y extraer la información enviada en el cuerpo de la solicitud.

Al construir una API RESTful que acepta datos en formato JSON, aplicaremos body-parser para analizar los datos de la solicitud JSON y extraer desde allí aquellos que necesitamos para procesar la misma.



Body Parser

Para implementar **body-Parser**, instalamos el mismo mediante **npm install body-parser**. Luego creamos una constante dentro del documento principal de nuestra aplicación Node.js y lo referenciamos.

```
● ● ● Body Parser  
  
//Terminal  
npm install body-parser  
  
//server.js  
const bodyParser = require('body-parser');
```

Luego, nos queda activar Body Parser para que comience a funcionar como Middleware dentro de nuestra aplicación Node.js, analizando cada dato que llegue en formato JSON.

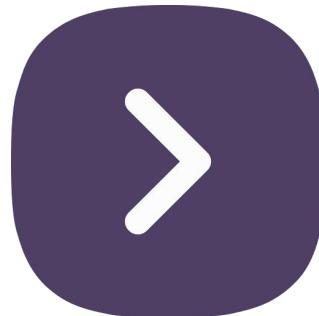
```
● ● ● Body Parser  
  
app.use(bodyParser.json());
```



El método next()

next() es una función middleware en Node.js la cual se ocupa de pasar el control de cada solicitud HTTP a la siguiente función de Middleware que exista en la pila.

Se integra dentro de **app.use()**, como un tercer parámetro relacionado a request y response. De no hacerlo, la solicitud queda “*colgada*”, sin enviar una respuesta al cliente.



El método next()

Definimos un nuevo Middleware previo a establecer las peticiones HTTP correspondientes al servidor web.

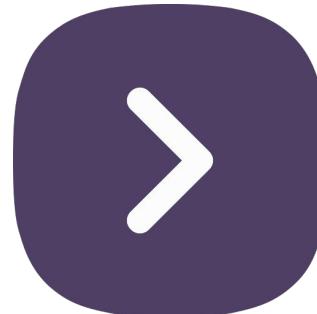
Junto a request y response se define a **next** como tercer parámetro. Luego de ejecutar el comando o tarea necesario invocamos a **next()**, ya como función, para que pase el control al resto de la lógica de nuestra aplicación.

```
...          next()  
  
app.use((req, res, next)=> {  
    //comando o acción a ejecutar.  
    next();  
})
```

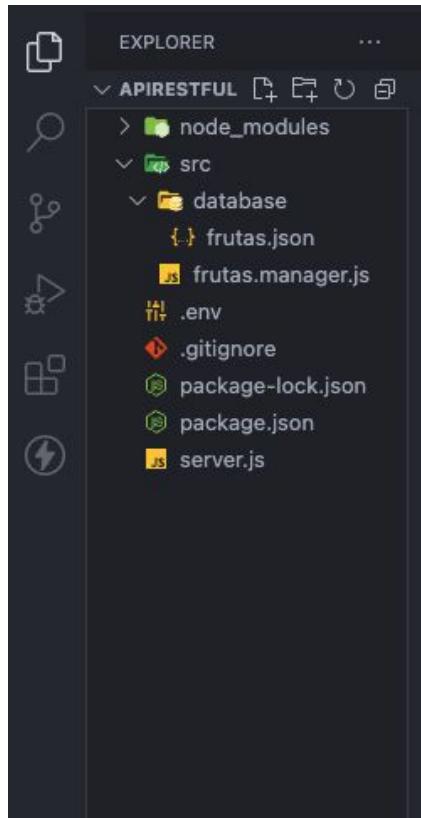
El método next()

Cada función middleware que tenga nuestra aplicación, se debe invocar a next para así pasar el control a la siguiente solicitud y que esta sea procesada correctamente.

De no hacerlo, la solicitud no pasará a la siguiente función, y la lógica de nuestra aplicación backend se detendrá en la función actual.



Proyecto modelo



Proyecto modelo

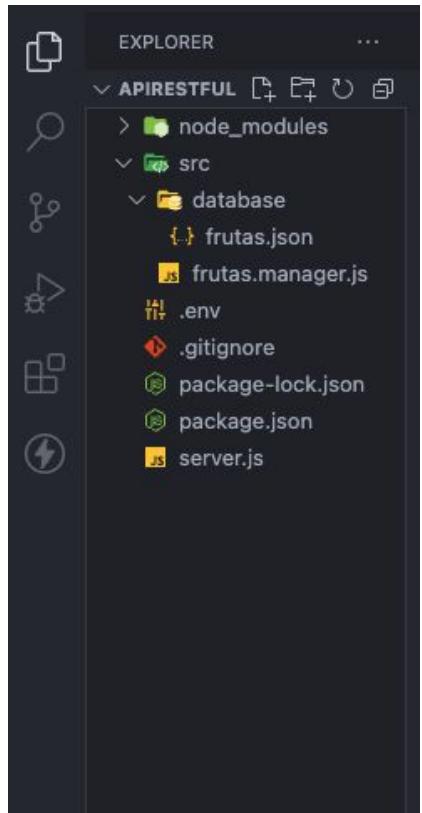
Elaboremos un proyecto modelo para poner en acción el uso de un archivo JSON con una estructura de array de objetos. Utilizaremos para ello nuestro ya conocido array: **frutas.json**.

El proyecto en cuestión tendrá un endpoint general, el cual usaremos para retornar todo el listado de frutas.

Y un segundo endpoint que escuchará un método **POST**, recibirá el recurso y lo guardará como nuevo recurso.

El archivo **frutas.json** será tratado como una base de datos, el cual será leído y escrito, cada vez que se cree un nuevo recurso.





Proyecto modelo

La estructura de nuestro proyecto contendrá una subcarpeta SRC, más una subcarpeta **/DATABASE** donde se guardará el archivo **frutas.json**.

Crearemos un archivo JS llamado **frutas.manager**, el cual dispondrá de dos funciones (*leer frutas y guardar frutas*) para manejar la información de nuestra base de datos.

Finalmente, el archivo **server.js** que tendrá la lógica de Express JS, integrando **frutas.manager** y **body-parser** junto a la lógica del servidor web.

La lógica del archivo frutas.manager.js

La lógica del archivo frutas.manager.js

```
frutas.manager.js

function guardarFrutas(frutas) {
  const datos = JSON.stringify(frutas);
  fs.writeFileSync(__dirname + process.env.DATABASE_PATH, datos);
}

module.exports = { leerFrutas, guardarFrutas };
```

La función **guardarFrutas()** recibirá como parámetro el array de objetos manipulado desde **server.js**, convertirá el contenido al formato de transporte de datos String para, finalmente, guardar este contenido en el archivo **frutas.json**.

Una vez creadas ambas funciones, exportamos las mismas utilizando **module.exports**.

La lógica del archivo frutas.manager.js

```
frutas.manager.js

function guardarFrutas(frutas) {
  const datos = JSON.stringify(frutas);
  fs.writeFileSync(__dirname + process.env.DATABASE_PATH, datos);
}

module.exports = { leerFrutas, guardarFrutas };
```

La función **guardarFrutas()** recibirá como parámetro el array de objetos manipulado desde **server.js**, convertirá el contenido al formato de transporte de datos String para, finalmente, guardar este contenido en el archivo **frutas.json**.

Una vez creadas ambas funciones, exportamos las mismas utilizando **module.exports**.

Variables de Entorno

Variables de Entorno

```
... .env  
PORT=3008  
DATABASE_PATH=/database/frutas.json
```

Nuestro archivo variable de entorno, manejará como siempre el puerto del servidor web, y además la ruta relativa hacia el archivo **frutas.json**.

La lógica del servidor web

La lógica del servidor web

```
server.js

const dotenv = require('dotenv');
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const { leerFrutas, guardarFrutas } = require('./src/frutas.manager');
const PORT = process.env.PORT || 3000;
let DB = [];
```

Declaramos
dotenv, express
y **body-parser**
en el header de
nuestro servidor.

Importamos también el archivo **frutas.manager** de su ruta relativa,
definimos el puerto y creamos la variable **DB**, que manejará los datos
del array de frutas dentro del servidor web.

La lógica del servidor web

```
● ● ● MIDDLEWARE

dotenv.config();

app.use(bodyParser.json());

app.use((req, res, next)=> {
    DB = leerFrutas();
    next();
})
```

Previo a poner a funcionar nuestro web server, declaramos todos los métodos y funciones correspondientes a la capa **Middleware**.

Activamos **bodyParser.json()** para que interprete todo contenido en este formato.

Por último, definimos el Middleware que obtendrá en la variable **DB** el contenido del archivo **frutas.json**.



La lógica del servidor web

```
● ● ● Servidor web

app.get('/', (req, res) => {
  res.send(DB);
});

//Aquí irá el método HTTP POST

app.get('*', (req, res) => {
  res.status(404).send('Lo siento, la
  página que buscas no existe.');
});
```

Definimos la respuesta de la ruta raíz del servidor, en la cual retornaremos la base de datos con todas las frutas en cuestión.

Al final del archivo, definimos el control de errores común, para toda ruta inexistente.

La lógica del servidor web

```
...  
Servidor web  
  
app.post('/', (req, res) => {  
  const nuevaFruta = req.body;  
  DB.push(nuevaFruta);  
  guardarFrutas(DB);  
  res.status(201).send('Fruta agregada!');  
});
```

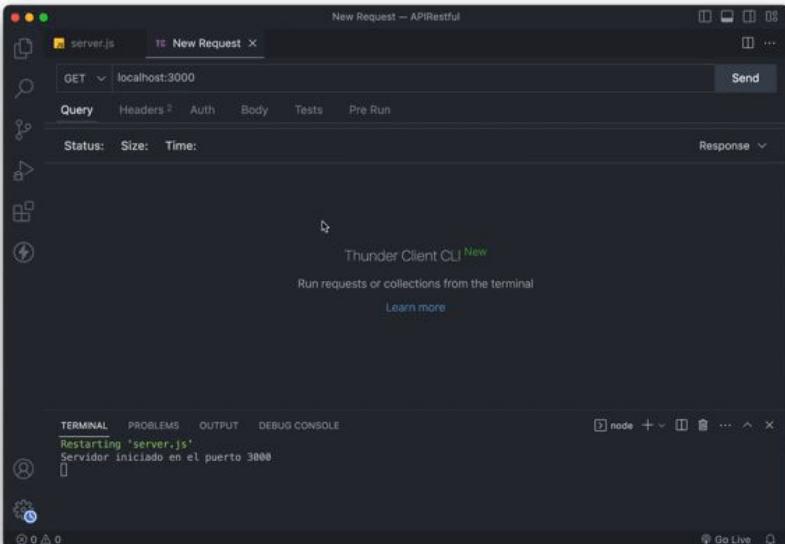
Finalmente invocamos el método **app.post()** el cual escucha la ruta raíz, a la espera de recibir un nuevo recurso.

La información del mismo la rescatamos de `request.body`, guardando la misma mediante el método **.push()** del array **DB**.

Por cada nuevo recurso recibido y guardado en el array, invocamos la función **guardarFrutas()** pasándole como parámetro el array DB.

¡Ignición!

Probamos la API Restful



Ya podemos utilizar Thunder Client para probar la petición GET simple de nuestro proyecto.

Como podemos apreciar, la lectura de la información del archivo **frutas.json** mediante **Filesystem API** funciona correctamente.

Probamos la API Restful

The screenshot shows a developer's workspace. At the top, a browser window displays a POST request to 'localhost:3000'. The response status is '200 OK', size is '1.08 KB', and time is '70 ms'. The response body contains JSON data representing a fruit object. Below the browser is a terminal window showing the command 'node server.js' being run, followed by the message 'Servidor iniciado en el puerto 3000'.

```
server.js  New Request -> APIRestful
GET localhost:3000
Query Headers 2 Auth Body Tests Pre Run
Status: 200 OK  Size: 1.08 KB  Time: 70 ms
Response
{
  "id": 16,
  "imagen": "●",
  "nombre": "Limonas",
  "importe": 260,
  "stock": 50
}
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE
Restarting 'server.js'
Servidor iniciado en el puerto 3000
Go Live
```

Y, por último, probamos la petición POST.

Recordemos tal como vimos con el ejemplo de JSON Placeholder, copiar la estructura de un objeto literal, pegarla en el cuerpo de la petición, modificando algunos valores previo a ejecutar el **método POST**.

Para validar su correcto guardado, podemos invocar nuevamente el método GET y revisar el último objeto del array. O, directamente, revisar el archivo **frutas.json** de nuestro proyecto web.



Prácticas

Prácticas

En base al proyecto que la profe utilizó en la clase de hoy, y que te compartirá como código base, deberás integrar los métodos HTTP faltantes para terminar de construir una API Restful completa.

Deberás agregar los métodos HTTP faltantes:

- PUT
- DELETE
- GET :id



Prácticas

PUT: este método HTTP permitirá modificar algún producto existente.

Debes agregar el parámetro **:id** en la URL para encontrar primero el producto, y luego modificarlo de acuerdo a los datos enviados en el cuerpo del mensaje.

Recuerda retornar un mensaje de error si no encuentra el producto.



DELETE: este método HTTP eliminará un producto existente.

También debes agregar el parámetro **:id** en la URL para identificar el producto en el array, y luego aplicar el método de array **.splice()**.

Recuerda retornar un mensaje de error si no encuentra el producto.

GET:id: este es el método convencional que nos permite ubicar un producto.

En este caso, aplica el método de array **.find()** para ubicar el producto y retornarlo.

Recuerda retornar un mensaje de error si no encuentra el producto.

Muchas gracias.



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

*primero
la gente*