**API en Express con métodos HTTP de tipo GET**

Después de haber entiendo cómo funciona Node.js en conjunto con Express, de analizar cómo funciona una API y los métodos HTTP más comunes, es hora de construir una pequeña aplicación de sólo lectura (con métodos GET), para que clientes como Angular puedan consumirla.

Implementación completa de la aplicación:

"use strict";

**const** express = require('express');

**const** app = express();

**const** port = 3000;

**const** host = '127.0.0.1';

**const** pendientesPorHacer = [

{id: 1, descripcion: "Comprar comida", completado: true},

{id: 2, descripcion: "Preparar comida", completado: false},

{id: 3, descripcion: "Pasear al perro", completado: true},

{id: 4, descripcion: "Estudiar React y VueJS", completado: false},

{id: 5, descripcion: "Terminar series pendientes", completado: false},

{id: 6, descripcion: "Reparar el celular", completado: true}

]

app.get('/', (req, res) **=>** {

return res.status(200).json(pendientesPorHacer);

})

app.get('/count', (req, res) **=>** {

**const** miRespuesta = {

"numeroDeRegistros": pendientesPorHacer.length

}

return res.status(200).json(miRespuesta);

})

.get('/:idPendiente', (req, res) **=>** {

**const** idPendiente = req.params.idPendiente;

**const** pendienteBuscado = pendientesPorHacer.find( pendiente => pendiente.id == idPendiente );

if( pendienteBuscado === undefined )

return res.status(404).json({ mensaje : "No se encontró ese pendiente en el servidor :(" });

else

return res.status(200).json(pendienteBuscado);

})

app.listen(port, host, () **=>** {

console.log(`¡Escuchando en ${host}:${port}/`);

});

Antes de ejecutar el código anterior, tendremos que instalar *express* localmente con: npm install express@4.16.2 --save

"use strict";

Como habrás notado, estamos implementando el modo estricto o “use strict”. Esto, en pocas palabras, le indica a JavaScript que sea menos permisivo con nuestro código al restringirnos algunas características de JavaScript que puede ser ineficientes al ser usadas incorrectamente. No es un dato del todo relevante, pero recomendamos que se utilice siempre que sea posible.

**const** express = require('express');

**const** app = express();

**const** port = 3000;

**const** host = '127.0.0.1';

Después viene la implementación de las variables globales que ya vimos en la introducción de Express. Como la importación del módulo de Express, la instanciación de Express en una variable global, llamada *app.*

También creamos dos constantes adicionales, una para el puerto o *port* y otro para el *host* de nuestra aplicación.

**const** pendientesPorHacer = [

{id: 1, descripcion: "Comprar comida", completado: true},

{id: 2, descripcion: "Preparar comida", completado: false},

{id: 3, descripcion: "Pasear al perro", completado: true},

{id: 4, descripcion: "Estudiar React y VueJS", completado: false},

{id: 5, descripcion: "Terminar series pendientes", completado: false},

{id: 6, descripcion: "Reparar el celular", completado: true}

]

Posteriormente, creamos un array que contendrá cada uno de los ***pendientes*** que usaremos para responderle a las diferentes peticiones (*requests*) de los clientes de nuestro servidor. Por efectos de simplicidad, en esta ocasión la información la escribimos directamente el código, ya que este ejercicio está dedicado a explicar cómo funciona una API con métodos HTTP de tipo GET. En posteriores ejercicios explicaremos cómo obtener la información desde una base de datos.

app.get('/', (req, res) **=>** {

return res.status(200).json(pendientesPorHacer);

})

Después con la instancia de *express* llamada *app*, establecemos que todas las peticiones HTTP de tipo GET ( app.get(...) ) que se dirijan a la ruta principal de la API ( '/' ) sean manejadas por una *callback* que les contestará con un *status code* de valor ‘200’ (es decir, un estado de respuesta exitoso) y el array de pendientesPorHacer convertido a formato JSON.

A modo de resumen, JSON o *JavaScript Object Notation* ofrece una manera de estructuración de datos en un formato ligero. Es usado principalmente para transmitir datos entre un servidor y una aplicación web, como una alternativa a *XML*.

*//Objeto de JavaScript*

{

id: 1,

descripcion: "Comprar comida",

completada: true,

funcion: **function** (){

*/\*Lógica de la función \*/*

}

}

*//JSON*

{

"id": 1,

"descripcion": "Comprar comida",

"completada": true,

"funcion": "No es posible poner funciones en un JSON"

}

Básicamente un JSON es un conjunto de datos en formato de *key/value* o *llave/valor.* Una *key* siempre tiene que ser una *string* entre comillas y *value* puede contener valores de tipo *string, number, boolean, array,* u *object*:

{

"key": "value",

"keyConNumberValue": 10,

"keyConBooleanValue": true,

"keyConArray": [],

"keyConObject": {

"otraKey": "otroValue"

}

}

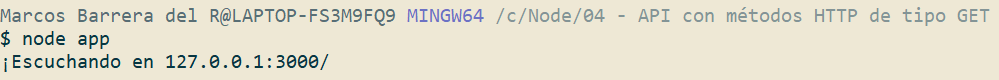
Continuando con la explicación de:

app.get('/', (req, res) **=>** {

return res.status(200).json(pendientesPorHacer);

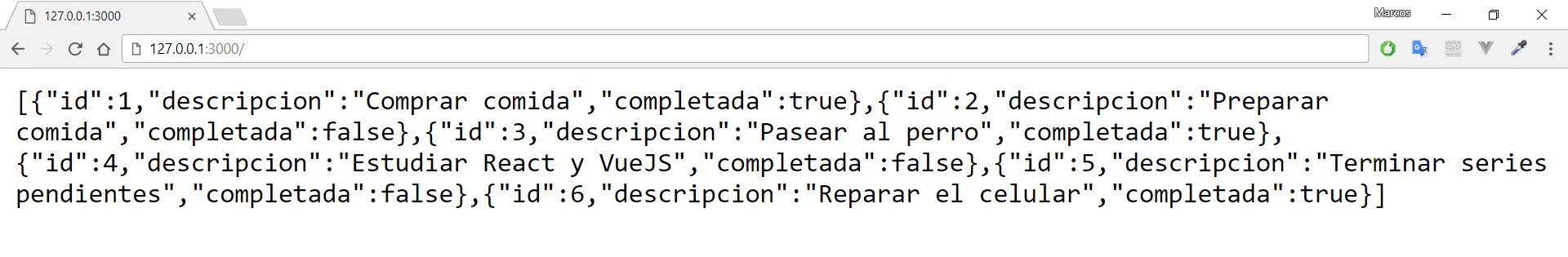
})

Para comprobar como funciona esta ruta, sencillamente tendremos que ingresar el siguiente comando en la consola: ***node <nombreDeNuestroArchivo>***



Y ya habremos puesto en marcha nuestro servidor. Así que nos podemos dirigir a nuestro navegador web e ingresar la URL de nuestro servidor con el puerto indicado, es decir, *127.0.0.1:3000*

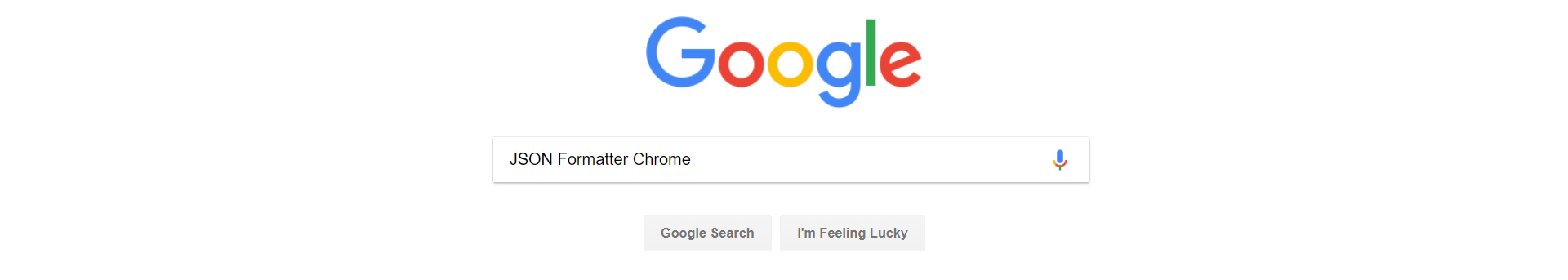
Una vez que ingresemos nos aparecerá lo siguiente:



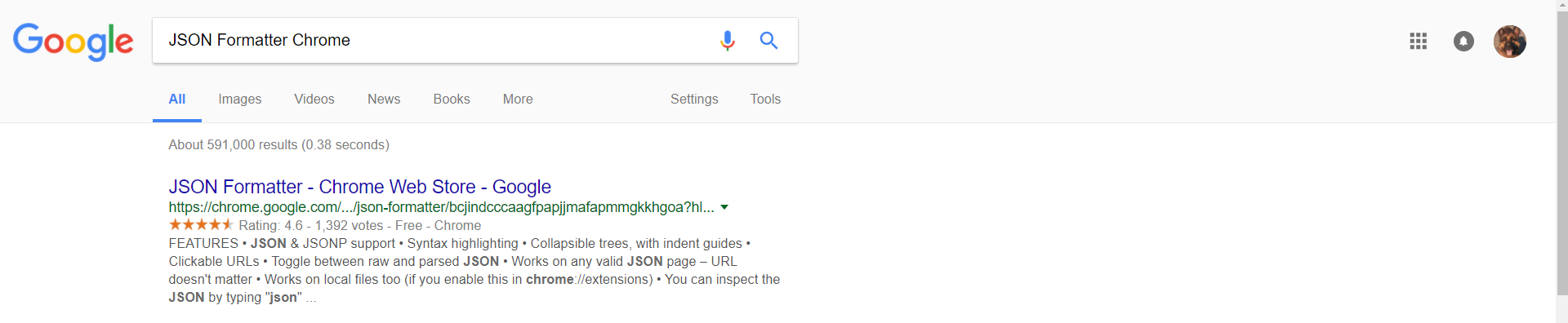
Como podemos ver, nuestro servidor de Node.js nos respondió con el array de pendientesPorHacer en formato JSON. Nuestro navegador web generó una petición de tipo GET a nuestro servidor en la ruta principal.

Y como se puede notar, la manera en la que se despliega la información es muy abultada y fea. Nosotros usamos Google Chrome y al igual que para muchos otros navegadores, hay extensiones que nos permiten darle un formato más adecuado a la manera que se despliegan los JSON en pantalla.

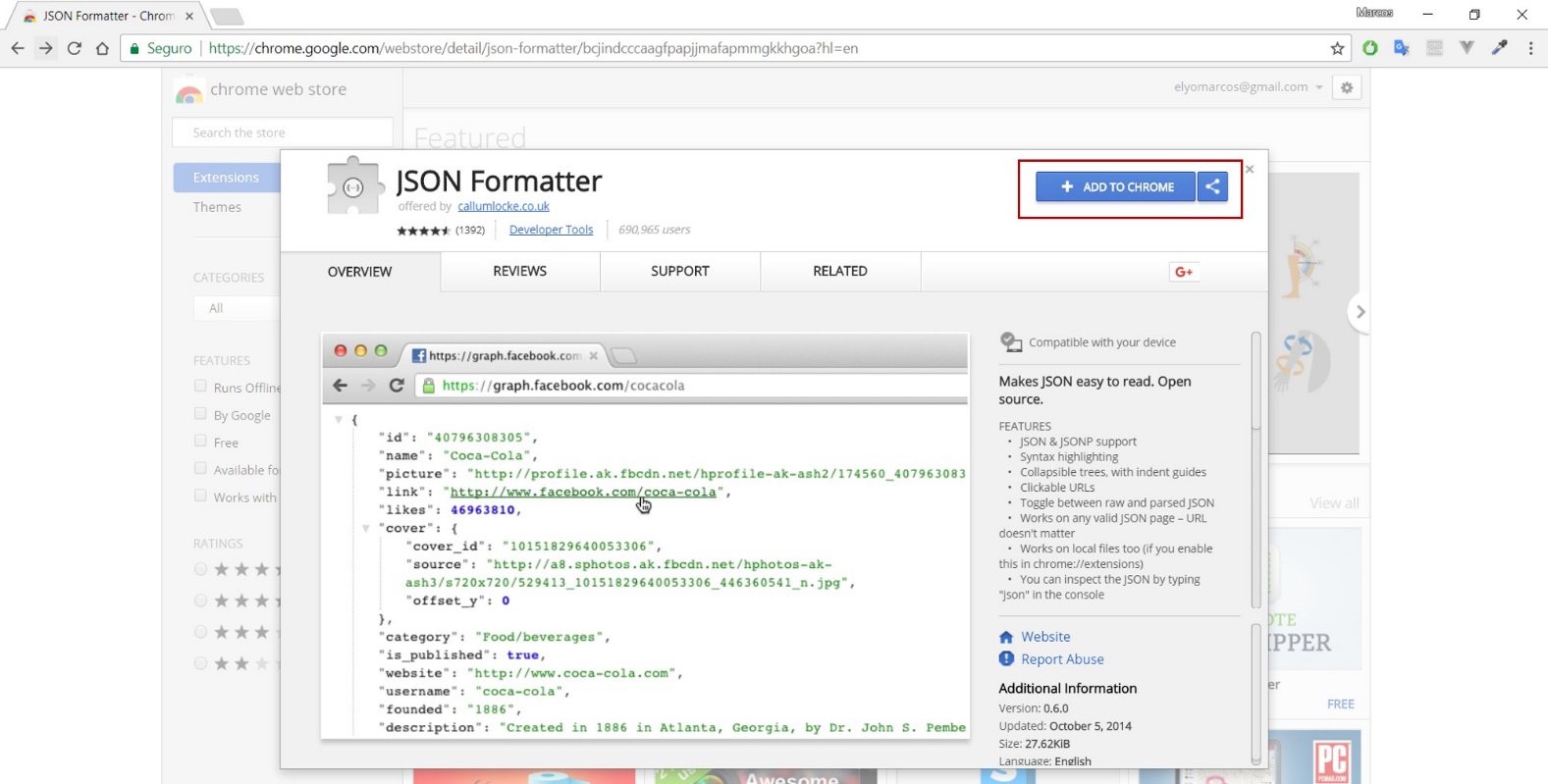
Basta con buscar en Google JSON Formatter Chrome:



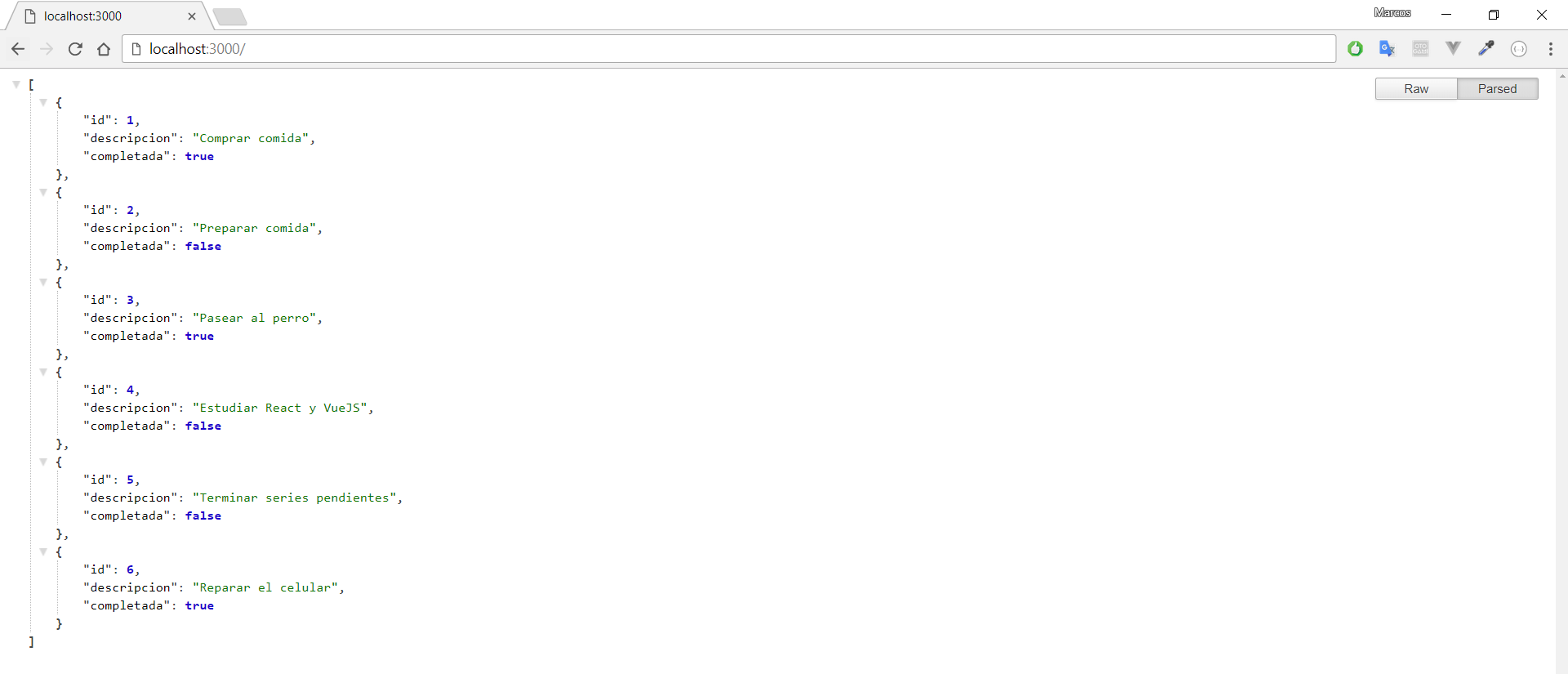
Seleccionamos la primera opción:



Y lo Añadimos a Chrome:



Una vez que tengamos añadida la extensión correctamente, si nos volvemos a dirigir a la URL de nuestro servidor “*127.0.0.1:3000*”, veremos nuestra respuesta JSON con un formato adecuado:



Después de haber probado la ruta principal de nuestro servidor, continuaremos con la siguiente ruta o *endpoint:*

app.get('/count', (req, res) **=>** {

**const** miRespuesta = {

"numeroDeRegistros": pendientesPorHacer.length

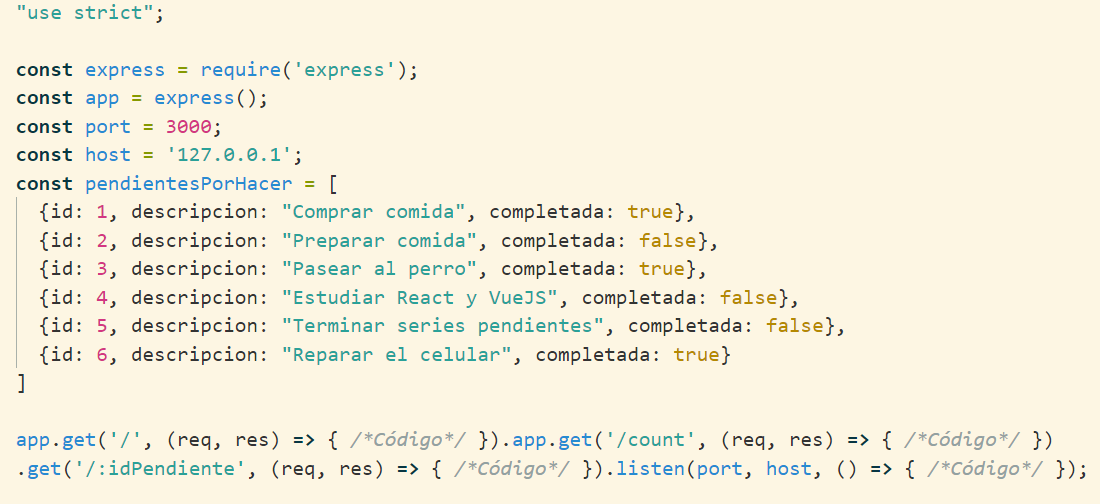
}

return res.status(200).json(miRespuesta);

})

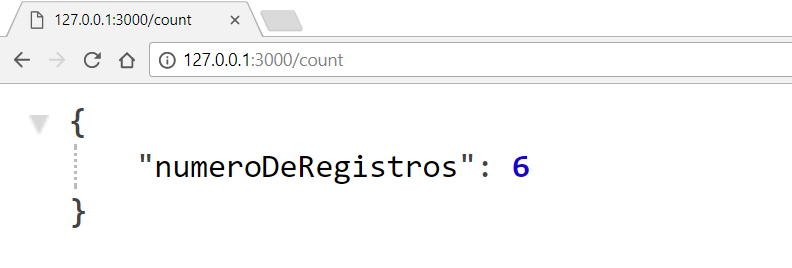
Si nos fijamos con detenimiento, podremos ver que hay un punto (.) justo al inicio de get(…). Esto significa que esta función sigue siendo llamada desde la instancia *app*, simplemente está encadenada con la anterior función “get('/', ...)”.

Otro acomodo del código sería el siguiente:



Podemos ver que cada una de las rutas (o *endpoints*) están encadenadas a la instancia *app* de *Express* y que incluso se puede encadenar el método/función .listen(...).

Bien, continuando con la explicación, el *endpoint*  “app.get('/count', (req, res) **=>** {...})”, le indica a *Express* que cada vez que llegue una petición (*request*) a nuestro servidor con el siguiente formato “127.0.0.1/count”, le responda al cliente con el número de registros que contiene nuestro de pendientesPorHacer.



Después, tenemos un *endpoint* con la siguiente estructura:

.get('/:idPendiente', (req, res) **=>** {

**const** idPendiente = req.params.idPendiente;

**const** pendienteBuscado = pendientesPorHacer.find( pendiente => pendiente.id == idPendiente );

if( pendienteBuscado === undefined )

return res.status(404).json({ mensaje : "No se encontró ese pendiente en el servidor :(" });

else

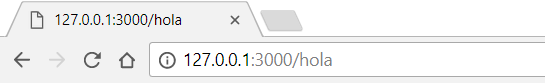
return res.status(200).json(pendienteBuscado);

})

Lo que le indicamos a *Express* con /:idPendiente es que idPendiente tendrá un valor dinámico de tipo *string* que encajará con el de la URL. Por ejemplo:



Al ingresar en la siguiente ruta, nuestro navegador le hará una petición de tipo GET a la ruta /:idPendiente y en este caso, idPendiente tomará el valor de “1”, ya que es lo que ingresamos en la URL.



En este otro caso, idPendiente tomaría el valor de “hola”.

**const** idPendiente = req.params.idPendiente;

Ahora, para poder obtener el valor de /:idPendiente , tendremos que utilizar al objeto *req*, el cual contiene otro objeto llamado *params.* Este último objeto contendrá todos los parámetros que hayamos establecido en la ruta.

En este caso, como nuestra ruta es /:idPendiente , este objeto sólo contendrá la propiedad llamada idPendiente. Es decir, un objeto como el siguiente:

{

idPendiente: "hola"

}

Si nuestra ruta tuviera múltiples parámetros, este objeto contendría el valor de los otros parámetros también. Por ejemplo, si tuviésemos la ruta /:idPendiente/:otroParametro' y si desde nuestro navegador la utilizáramos de la siguiente manera:



Nuestro objeto *req.params* tendría los siguiente valores:

{

idPendiente: "hola",

otroParametro: "mivalordelotroparametro",

}

Así que si ingresamos la URL de “*127.0.0.1:3000/5*” en nuestro navegador web, este hará una petición GET a la ruta /:idPendiente.

.get('/:idPendiente', (req, res) **=>** {

**const** idPendiente = req.params.idPendiente;

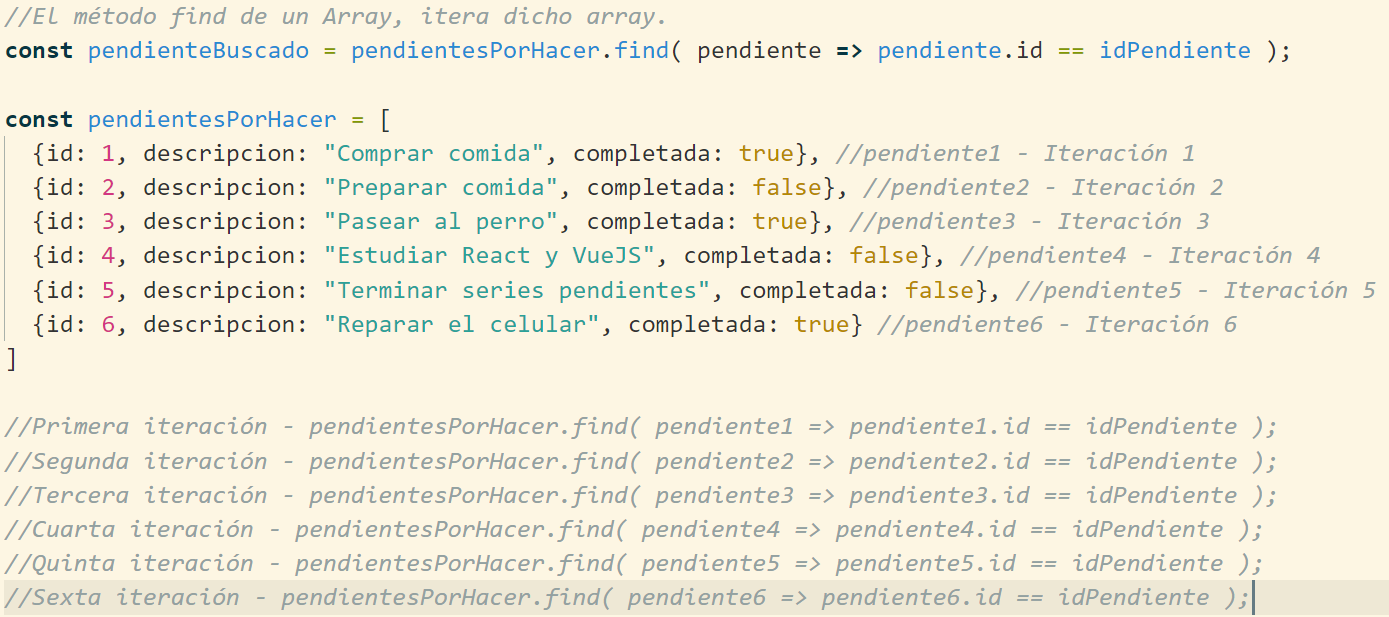
*// ...*

})

Y la constante idPendiente, tomará el valor de 5, ya que, como explicamos anteriormente, el objeto req.params contiene la de .idPendiente en él.

**const** pendienteBuscado = pendientesPorHacer.find( pendiente => pendiente.id == idPendiente );

Después, en el array de pendientesPorHacer, buscaremos con el método find, y por cada pendiente almacenado en el array, find hará una iteración en el array comparando la constante idPendiente, con pendiente.id de cada uno de los pendientes. Una representación sería la siguiente:



El método find hará las iteraciones hasta que la siguiente comparación sea **true**:

pendiente.id == idPendiente

Y cuando dicha comparación sea **true**, el método find retornará el objeto con el *id* que haya coincidido con la constante idPendiente.

Es decir, en nuestro caso, como idPendiente vale 5, nuestro objeto con *id* 5 será:

{id: 5, descripcion: "Terminar series pendientes", completada: false}

Así que find retornará ese objeto y dicho objeto será asignado a la constante pendienteBuscado. Al haber encontrado una coincidencia, find detendrá las iteraciones y no hará la iteración del sexto elemento.

**Nota:**  El método ***find*** puede ser utilizado en todos los arrays a partir de la introducción de ECMAScript 6. Para más información sobre ***find***, consulte el siguiente enlace.

<https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/find>

//. . .

**const** pendienteBuscado = pendientesPorHacer.find( pendiente => pendiente.id == idPendiente)

if( pendienteBuscado === undefined )

return res.status(404).json({ mensaje : "No se encontró ese pendiente en el servidor :(" });

else

return res.status(200).json(pendienteBuscado);

})

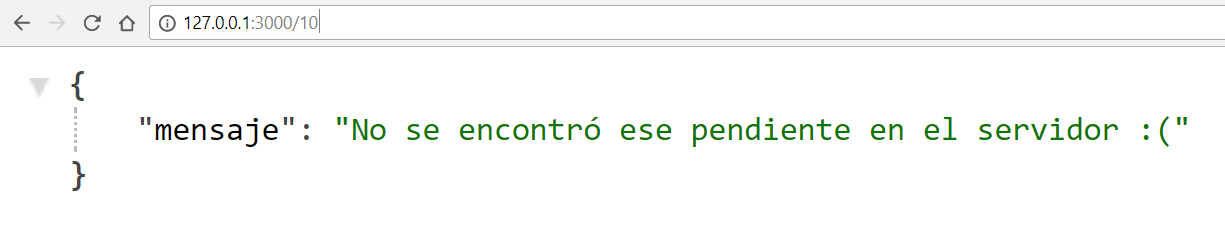
Finalmente, después de haber hecho la búsqueda con el método find, utilizamos una sentencia if/else y decimos que, si pendienteBuscado es igual *undefined*, responderemos con un *statusCode* “404” y un mensaje de diciendo que no se encontró ese pendiente en el servidor.

Esto lo hacemos, ya que si el método find no encuentra ningún elemento en el array la comparación que explicamos anteriormente entonces regresará *undefined.*

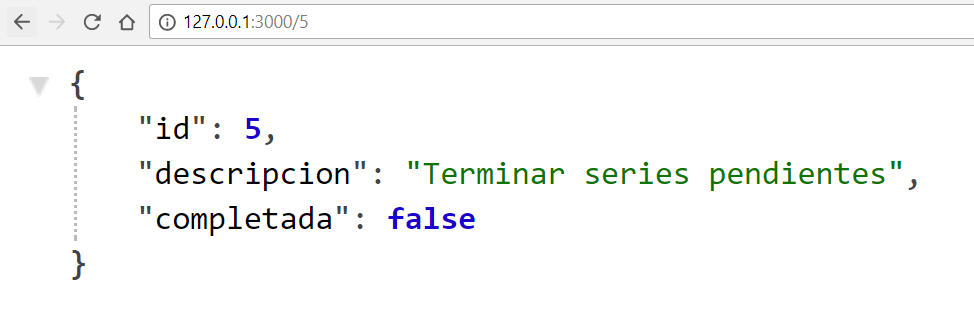
Por ejemplo, si la URL que utilizásemos en nuestro navegador, fuera la siguiente:

*127.0.0.1:3000/10*

Eso haría que el método find quisiese encontrar un elemento con *id* de valor “10”, y como en nuestro array de pendientesPorHacerno tenemos ningún pendiente con ese id, find no encontraría nada y por lo tanto retornaría *undefined* y esto sería asignado a pendienteBuscado. Esto significaría que no encontramos tal registro en el array y por lo tanto retornamos un mensaje de no encontrado al cliente.



Por el contrario, si ingresáramos un id en la URL que sí coincidiera con el id de algún elemento en nuestro array de pendientesPorHacer, eso ejecutaría la sentencia else, y le responderíamos al cliente con un *statusCode* “200” y con el elemento que solicitó.



Y finalmente, quedaría esta última porción de código:

app.listen(port, host, () **=>** {

console.log(`¡Escuchando en ${host}:${port}/`);

});

Esto le indica a *Express* que se ponga a la escucha por el puerto y host especificados, es decir, por el puerto *3000* y con el host *127.0.0.1.* Una vez que *Express* se pone a la espera de peticiones, se imprimirá en la consola “ *¡Escuchando en 127.0.0.1:3000/ ”*.