**API en Express con CRUD Básico**

Ahora que sabemos cómo hacer registros en nuestro servidor mediante el método POST de HTTP, es hora de implementar un CRUD básico, por sus siglas en inglés: ***C****reate,* ***R****ead,* ***U****pdate and* ***D****elete* (en español, *Crear, Leer, Actualizar y Eliminar).*

Para poder implementar dicha funcionalidad, tendremos que utilizar los métodos HTTP anteriormente vistos (GET, POST) y dos nuevos métodos HTTP llamados PUT y DELETE, de los cuales ya hemos explicado su funcionalidad en secciones anteriores.

Nuevamente, deberemos instalar *Express* si aún no lo tenemos instalado con el comando “***npm install express@4.16.2 --save”*** al igual que Body Parser, con “***npm install body-parser@1.18.2 --save***”.

Una que tengamos instalado *Express* y *body-parser*, crearemos un archivo llamado “*app.js”* como es habitual. El código completo de la aplicación es el siguiente:

"use strict";

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

**const** app = express();

**const** bodyParser = require('body-parser');

app.use(bodyParser.json());

**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}

]

*//Función para encontrar el siguiente Id.*

**function** obtenerSiguienteId() {

**const** tamañoArray = pendientesPorHacer.length;

**let** idMayor = pendientesPorHacer[0].id || 1;

for (**let** i = 1; i < tamañoArray; i++) {

if(pendientesPorHacer[i].id > idMayor)

idMayor = pendientesPorHacer[i].id;

}

return idMayor + 1;

}

**function** encontrarPorId( idPendiente ) {

**const** pendienteIndex = pendientesPorHacer.findIndex( pendiente **=>** pendiente.id == idPendiente );

*//Si se encuentra el pendiente, 'pendienteBuscado' contendrá*

*//la posición del pendiente en el array de 'pendientesPorHacer'*

*//sino, 'pendienteBuscado' contrendrá una -1*

return pendienteIndex;

}

*//Consulta general*

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

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

})

*//Consultar cuando registros hay en el array de pendientesPorHacer.*

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

**const** miRespuesta = {

"numeroDeRegistros": pendientesPorHacer.length

}

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

})

*//Consultar un elemento en el array de pendientesPorHacer por Id.*

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

**const** pendienteIndex = encontrarPorId( req.params.idPendiente );

if( pendienteIndex === -1 )

return res.status(404).json({

mensaje : "No se encontró ese pendiente en el servidor :(" });

else

return res.status(200).json( pendientesPorHacer[pendienteIndex] );

})

*//Agregar un nuevo elemento al array.*

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

*//Body Parser nos permite obtener los parámetros de las peticiones POST*

*//en el objeto req.body*

**const** nuevoPendiente = {

id: obtenerSiguienteId(),

descripcion: req.body.descripcion,

completado: req.body.completado

}

pendientesPorHacer.push(nuevoPendiente);

**const** miRespuesta = {

registroAgregado: nuevoPendiente,

mensaje: "Registro agregado exitosamente :)"

}

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

})

*//Modifica un elemento existente en el al array.*

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

**const** pendienteIndex = encontrarPorId( req.params.idPendiente );

if( pendienteIndex === -1 ) {

return res.status(404).json({

mensaje : "No se encontró ese pendiente en el servidor :(" });

}

*//Body Parser nos permite obtener los parámetros de las peticiones PUT*

*//en el objeto req.body*

**const** pendienteEditado = {

id: pendientesPorHacer[ pendienteIndex ].id,

descripcion: req.body.descripcion || pendientesPorHacer[ pendienteIndex ].descripcion,

completado: req.body.completado !== undefined

? req.body.completado

: pendientesPorHacer[ pendienteIndex ].completado

}

*//Reemplazamos el pendiente actual por el modificado.*

pendientesPorHacer[ pendienteIndex ] = pendienteEditado;

**const** miRespuesta = {

registroModificado: pendienteEditado,

mensaje: "Registro editado exitosamente :)"

}

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

})

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

**const** pendienteIndex = encontrarPorId( req.params.idPendiente );

if( pendienteIndex === -1 ) {

return res.status(404).json({

mensaje : "No se encontró ese pendiente en el servidor :(" });

}

**const** pendienteEliminado = pendientesPorHacer[ pendienteIndex ];

*//Eliminamos el pendiente del array 'pendientesPorHacer'*

*//con Array.splice.*

pendientesPorHacer.splice( pendienteIndex, 1 );

**const** miRespuesta = {

registroEliminado: pendienteEliminado,

mensaje: "Registro eliminado exitosamente cx"

}

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

})

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

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

});

Como puedes ver, modificamos el ejercicio anterior y agregamos los métodos “*.put(…)”* y “*.delete(…)”* al servidor, con el fin de poder modificar y eliminar los pendientes. Además, corregimos un pequeño error gramatical y ahora la propiedad “*completada*” de los pendientes fue cambiada a “*completado”.*

Como ya hemos explicado muchas partes que conforman este código en anteriores ejercicios, sólo explicaremos los nuevos cambios que hemos implementado.

El primer cambio fue que hicimos una nueva función llamada encontrarPorId:

**function** encontrarPorId( idPendiente ) {

**const** pendienteIndex = pendientesPorHacer.findIndex( pendiente **=>** pendiente.id == idPendiente );

*//Si se encuentra el pendiente, 'pendienteBuscado' contendrá*

*//la posición del pendiente en el array de 'pendientesPorHacer'*

*//sino, 'pendienteBuscado' contrendrá una -1*

return pendienteIndex;

}

Y nos permite buscar el índice de un pendiente, con base en un parámetro llamada *idPendiente.* Si miramos con detenimiento, es la misma búsqueda que teníamos anteriormente en la consulta individual (es decir, en “.get('/:idPendiente', (req, res) **=>** {...})”), pero con la pequeña diferencia de que aquí utilizamos el método findIndex() de los arrays, en vez del método find(). La única diferencia entre estos dos métodos es que findIndex() retorna el *index* (la posición) del elemento buscado, si es que lo encuentra (find() retorna el elemento en lugar del *index*). En caso de que no lo encuentre, findIndex() retorna el número -1 (find() retorna *undefined*).

El siguiente cambio que introdujimos fue que en la ruta “.get('/:idPendiente', (req, res) **=>** {...}” utilizamos la función “encontrarPorId( idPendiente )”:

*//Consultar un elemento en el array de pendientesPorHacer por Id.*

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

**const** pendienteIndex = encontrarPorId( req.params.idPendiente );

if( pendienteIndex === -1 )

return res.status(404).json({

mensaje : "No se encontró ese pendiente en el servidor :(" });

else

return res.status(200).json( pendientesPorHacer[pendienteIndex] );

})

Y verificamos que pendienteIndex no contenga el valor de *-1*. En caso de que contenga *-1 (*es decir, que no haya encontrado el *pendiente* con el *id* dado), le responderemos al cliente con un mensaje de error. En cambio, si pendienteIndex no vale *-1,* significa que contiene el *index* del elemento que buscamos, y por lo tanto, responderemos con el elemento del array en el *index* encontrado.

Después, implementamos el código para poder editar pendientes que ya se encuentren en el servidor:

*//Modifica un elemento existente en el array.*

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

**const** pendienteIndex = encontrarPorId( req.params.idPendiente );

if( pendienteIndex === -1 ) {

return res.status(404).json({

mensaje : "No se encontró ese pendiente en el servidor :(" });

}

**const** pendienteEditado = {

id: pendientesPorHacer[ pendienteIndex ].id,

descripcion: req.body.descripcion || pendientesPorHacer[ pendienteIndex ].descripcion,

completado: req.body.completado !== undefined

? req.body.completado

: pendientesPorHacer[ pendienteIndex ].completado

}

*//Reemplazamos el pendiente actual por el modificado.*

pendientesPorHacer[ pendienteIndex ] = pendienteEditado;

**const** miRespuesta = {

registroModificado: pendienteEditado,

mensaje: "Registro editado exitosamente :)"

}

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

})

Como podemos ver, el método ahora es “*.put(…)”*. Como ya vimos en anteriores secciones, el método PUT de HTTP nos permite indicarle a un servidor que la petición se dirige a un *endpoint* de modificación. Aunque “*.put(…)”* le indica a *Express* que este *endpoint* es para modificar uno o varios registro en el servidor, técnicamente se puede aplicar cualquier lógica que queramos en dicho *endpoint,* es decir, podríamos consultar, agregar o incluso eliminar en un endpoint de tipo “.*put(…)”*. Sin embargo, no es recomendable. Lo que se recomienda hacer es respetar cada uno de los verbos HTTP y que cada *endpoint* que programemos, se apegue a las definiciones del verbo HTTP que se utilice.

Es decir, GET para consultar, PUT y PATCH para modificar, DELETE para eliminar y POST para agregar o para aplicar cualquier otra lógica que no encaje en los otros métodos de tipo HTTP.

Entonces, lo primero que hacemos es tratar de obtener el *index* del elemento deseado con el *id* extraído de la ruta del método “*.put(…)*”:

**const** pendienteIndex = encontrarPorId( req.params.idPendiente );

Enseguida, verificamos si el *index* retornado por la función encontrarPorId() es válido, sino lo es, le respondemos al cliente con un mensaje que indica que no pudimos encontrar dicho *pendiente* en el servidor.

Después, si pendienteIndex contiene un *index* válido, contruimos un objeto llamado pendienteEditado que contendrá el mismo *id* del *pendiente* que queremos actualizar, la *descripción* que venga en la petición (*request*) y el estado de *completado* que venga en la petición. En caso de que o la *descripción* o el estado de *completado* no vengan en la petición, utilizaremos el valor que ya tenía anteriormente el *pendiente* que está siendo modificado.

**const** pendienteEditado = {

id: pendientesPorHacer[ pendienteIndex ].id,

descripcion: req.body.descripcion || pendientesPorHacer[ pendienteIndex ].descripcion,

completado: req.body.completado !== undefined

? req.body.completado

: pendientesPorHacer[ pendienteIndex ].completado

}

descripcion: req.body.descripcion || pendientesPorHacer[ pendienteIndex ].descripcion

La propiedad de *descripcion* tomará el valor de req.body.descripcion en caso de que exista, es decir, en caso de que el cliente haya mandado ese parámetro en el body de la *request*. En caso de que el cliente no haya mandado descripcion en la *request*, la propiedad de *descripcion* tomará el valor de “pendientesPorHacer[pendienteIndex].descripcion”, es decir, el valor del *pendiente* que se está modificando actualmente.

completado: req.body.completado !== undefined

? req.body.completado

: pendientesPorHacer[ pendienteIndex ].completado

En el caso de los valores *booleanos,* utilizaremos una técnica de asignación con ternarios. Los ternarios son como una sentencia *if/else*.

Si *completado* es no es *undefined*, es decir, el usuario lo mandó en la *request.*

req.body.completado !== undefined

Entonces asígnale el valor de req.body.completado a la propiedad completado

Si *completado* es *undefined,* entonces haz que completado valga lo que actualmente tiene el *pendiente* del array, es decir, que valga pendientesPorHacer[ pendienteIndex ].completado.

Otra manera de interpretarlo sería la siguiente:

req.body.completado !== undefined

? req.body.completado

: pendientesPorHacer[ pendienteIndex ].completado

if( req.body.completado !== undefined ) {

*//Si el valor de 'completado' fue mandado por el cliente,*

*//se asignará lo que envió el cliente:*

pendienteEditado.completado = req.body.completado;

} else {

*//Si el valor de 'completado' no fue mandado por el cliente*

*//se asignará el valor del pendiente actual antes de hacer*

*//la modificación:*

pendienteEditado.completado = pendientesPorHacer[ pendienteIndex ].completado;

}

Pero para evitarnos todas esas líneas de código, preferimos utilizar la sintaxis de operadores ternarios.

Bien… después, remplazamos el *pendiente* que esté en la posición de pendienteIndex (es decir, el *pendiente* que el cliente quiere modificar) y lo reemplazamos por el *pendiente* que construimos, llamado pendienteEditado.

pendientesPorHacer[ pendienteIndex ] = pendienteEditado;

Por último en el *endpoint* de “*.put(...)*”, construimos una respuesta que contenga el registro modificado y un mensaje de éxito. Y retornamos dicha respuesta al cliente con un *status* 200 (*status* exitoso) y nuestra respuesta en formato JSON.

**const** miRespuesta = {

registroModificado: pendienteEditado,

mensaje: "Registro editado exitosamente :)"

}

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

Finalmente, el último cambio que hicimos fue agregar el *endpoint* para el verbo de HTTP llamado DELETE; el cual nos indica que se pretende eliminar un registro en el servidor.

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

**const** pendienteIndex = encontrarPorId( req.params.idPendiente );

if( pendienteIndex === -1 ) {

return res.status(404).json({

mensaje : "No se encontró ese pendiente en el servidor :(" });

}

**const** pendienteEliminado = pendientesPorHacer[ pendienteIndex ];

*//Eliminamos el pendiente del array 'pendientesPorHacer'*

*//con Array.splice.*

pendientesPorHacer.splice( pendienteIndex, 1 )

**const** miRespuesta = {

registroModificado: pendienteEliminado,

mensaje: "Registro editado exitosamente :)"

}

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

})

Al igual que en el *endpoint de “*.get('/:idPendiente',(req, res)**=>**{...}*”* y *“*.put('/:idPendiente',(req, res)**=>**{...}*”,* en este *endpoint* obtenemos el *idPendiente* que el cliente manda desde mediante la URL, después buscamos que exista algún pendiente con dicho *idPendiente* con la función encontrarPorId(), y guardamos lo que retorna dicha función en una constante llamada pendienteIndex, al igual que en los otros *enpoint mencionados.*

**const** pendienteIndex = encontrarPorId( req.params.idPendiente );

Después verificamos si el *index* es válido, sino lo es, le informamos al cliente que hubo un error. En caso de que el *index* sea válido, guardamos una referencia al *pendiente* que está por eliminarse en una constante llamada pendienteEliminado, para poder informarle al cliente qué registro eliminó. Y con el método splice() que incorporan los array de JavaScript, eliminamos el *pendiente* deseado con el pendienteIndex. Y por último le respondemos al cliente con un mensaje que contiene el *pendiente* eliminado y un mensaje.

if( pendienteIndex === -1 ) {

return res.status(404).json({

mensaje : "No se encontró ese pendiente en el servidor :(" });

}

**const** pendienteEliminado = pendientesPorHacer[ pendienteIndex ];

*//Eliminamos el pendiente del array 'pendientesPorHacer'*

*//con Array.splice.*

pendientesPorHacer.splice( pendienteIndex, 1 )

**const** miRespuesta = {

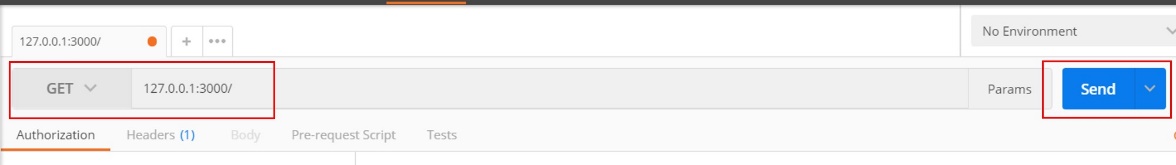
registroEliminado: pendienteEliminado,

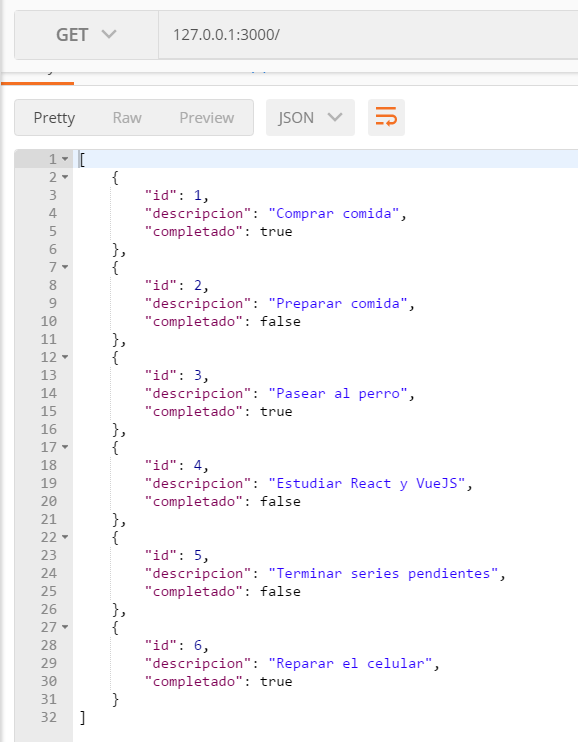
mensaje: "Registro eliminado exitosamente cx"

}

Bien, ahora que ya explicamos cómo funciona el código nuevo que implementamos, es hora de probar los nuevos *endpoints.*

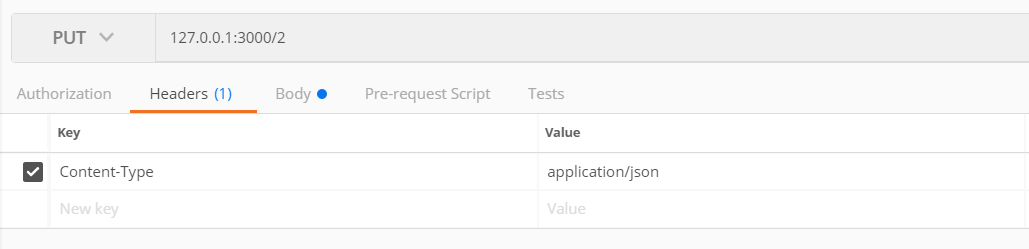
Primero correremos nuestro servidor con el comando de ***nodemon app.js*** y abriremos Postman. Una vez allí, con el método GET haremos una *request* a nuestro servidor con la URL de “*127.0.0.1:3000/*” y daremos clic en “*SEND*”:



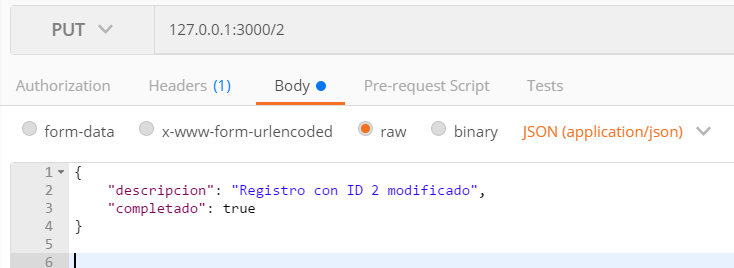


Como respuesta podemos ver los *pendientes* actuales que se encuentran en nuestro array de *pendientePorHacer.*

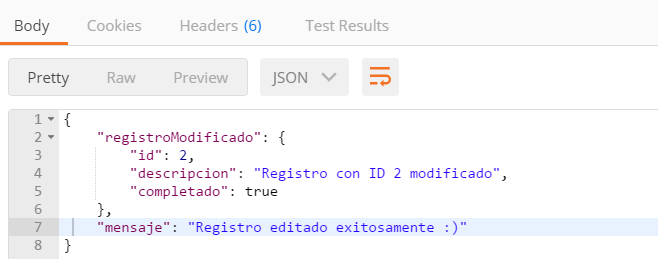
Ahora, para probar el nuevo *endpoint* para modificar un registro, cambiaremos la URL a “*127.0.0.1:3000/2*”, seleccionaremos PUT, después en la pestaña de Headers, como *Key* escribiremos “*Content-Type*” y en su *Value* escribiremos “*application/json*” como vimos en el Ejercicio 04.



En seguida, no dirigiremos a la pestaña de *Body* y agregaremos los nuevos valores del registro que deseemos modificar:



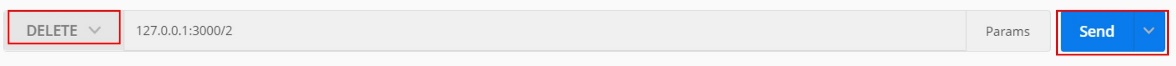
Cuando hayamos acabado de agregar nuestros campos a modificar, daremos clic en “*SEND*” y *Express* nos responderá con algo similar a esto:



Una vez que hayamos hecho esto, podemos verificar en la consulta general que el cambio se haya realizado:



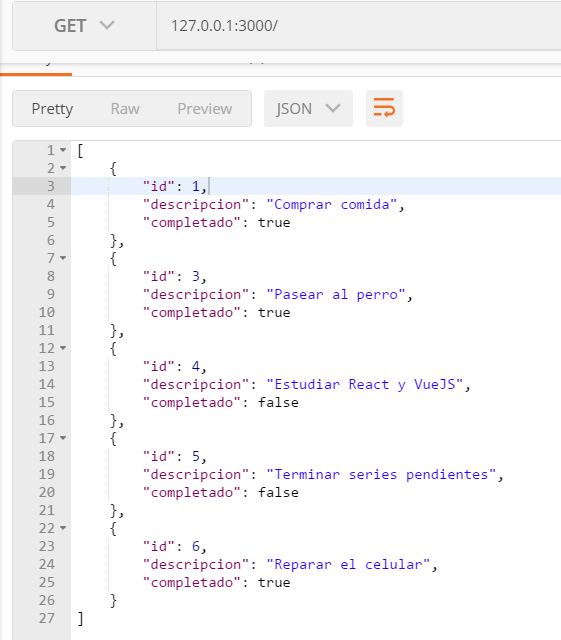
Ahora, para probar la eliminación de nuestra API, eliminemos el *pendiente* con ID 2, es decir, el que modificamos anteriormente. Para ello, tendremos que seleccionar el método DELETE, ingresar la siguiente URL “*127.0.0.1:3000/2*” y dar clic en “*SEND*”:



*Express* responderá con lo siguiente:



Lo que significa que el *pendiente* se ha eliminado debidamente. Para comprobarlo, haremos una última consulta general:



Y como podemos ver, el registro con *ID* de valor de *2* ya no se encuentra disponible.