

Node.js

en el desarrollo de aplicaciones web y servicios

4. El framework Express (I) Conceptos básicos

Ignacio Iborra Baeza

Índice de contenidos

Node.js	1
1. Introducción. Servicios REST	3
1.1. La base: el protocolo HTTP y las URL	3
1.2. Los servicios REST	3
1.3. El formato JSON	4
1.3.1. Conversión a y desde JSON	5
2. ¿Qué es Express?	6
2.1. Descarga e instalación	6
2.1.1. Ejemplo de servidor básico con Express	6
2.2. Express como proveedor de servicios	7
2.2.1. Un primer servicio básico	7
2.3. Elementos básicos: aplicación, petición y respuesta	8
2.3.1. La aplicación	8
2.3.2. La petición	8
2.3.3. La respuesta	9
3. Ejemplo de enrutamiento simple	10
3.1. Esqueleto básico del servidor principal	10
3.1.1. Servicios a desarrollar	11
3.2. Servicios de listado (GET)	11
3.2.1. Listado de todos los contactos	11
3.2.2. Ficha de un contacto a partir de su id	11
3.2.3. Uso de la query string para pasar parámetros	12
3.2.4. Prueba de los servicios	12
3.3. Servicios de inserción, borrado y modificación	13
3.3.1. Inserción de contactos (POST)	13
3.3.2. Modificación de contactos (PUT)	14
3.3.3. Más sobre "body-parser"	14
3.3.4. Borrado de contactos (DELETE)	15
3.3.5. Sobre el resultado de la actualización o el borrado	15
4. Probar los servicios con Postman	16
4.1. Descarga, instalación y primeros pasos	16
4.2. Añadir peticiones simples: GET	18
4.3. Añadir peticiones POST	20
4.4. Añadir peticiones PUT o DELETE	21
4.5. Exportar/Importar colecciones	21
5. Ejercicios	23
5.1. Ejercicio 1	23
5.2. Ejercicio 2	23
5.3. Ejercicio 3	23

1. Introducción. Servicios REST

1.1. La base: el protocolo HTTP y las URL

Para lo que vamos a ver a partir de esta sesión, conviene tener claros algunos conceptos de base. Para empezar, cualquier aplicación web se basa en una arquitectura cliente-servidor, donde un servidor queda a la espera de conexiones de clientes, y los clientes se conectan a los servidores para solicitar ciertos recursos.

Estas comunicaciones se realizan mediante un protocolo llamado **HTTP** (o HTTPS, en el caso de comunicaciones seguras). En ambos casos, cliente y servidor se envían cierta información estándar, en cada mensaje:

- En cuanto a los clientes, envían al servidor los datos del recurso que solicitan, junto con cierta información adicional, como por ejemplo las cabeceras de petición (información relativa al tipo de cliente o navegador, contenido que acepta, etc), y parámetros adicionales llamados normalmente *datos del formulario*.
- Por lo que respecta a los servidores, aceptan estas peticiones, las procesan y envían de vuelta algunos datos relevantes, como un código de estado (indicando si la petición pudo ser atendida satisfactoriamente o no), cabeceras de respuesta (indicando el tipo de contenido enviado, tamaño, idioma, etc), y el recurso solicitado propiamente dicho, si todo ha ido correctamente.

Para solicitar los recursos, los clientes se conectan o solicitan una determinada **URL** (siglas en inglés de "localización uniforme de recursos", *Uniform Resource Location*). Esta URL consiste en un fragmento de texto con tres secciones diferenciadas:

- El protocolo empleado (HTTP o HTTPS)
- El nombre de dominio, que identifica al servidor y lo localiza en la red.
- La ruta hacia el recurso solicitado, dentro del propio servidor. Esta última parte también suele denominarse URI (identificador uniforme de recurso, o en inglés, *Uniform Resource Identifier*). Esta URI identifica unívocamente el recurso buscado entre todos los demás recursos que pueda albergar el servidor.

Por ejemplo, la siguiente podría ser una URL válida:

`http://miservidor.com/libros?id=123`

El protocolo empleado es *http*, y el nombre de dominio es *miservidor.com*. Finalmente, la ruta o URI es *libros?id=123*, y el texto tras el interrogante '?' es la información adicional llamada *datos del formulario*. Esta información permite aportar algo más de información sobre el recurso solicitado. En este caso, podría hacer referencia al código del libro que estamos buscando.

1.2. Los servicios REST

En esta sesión del tema veremos cómo aplicar lo aprendido hasta ahora para desarrollar un servidor sencillo que proporcione una API REST a los clientes que se conecten. REST son las siglas de *REpresentational State Transfer*, y designa un estilo de arquitectura de aplicaciones distribuidas, como las aplicaciones web. En un sistema REST, identificamos

cada recurso a solicitar con una URI (identificador uniforme de recurso), y definimos un conjunto delimitado de comandos o métodos a realizar, que típicamente son:

- GET: para obtener resultados de algún tipo (listados completos o filtrados por alguna condición)
- POST: para realizar inserciones o añadir elementos en un conjunto de datos
- PUT: para realizar modificaciones o actualizaciones del conjunto de datos
- DELETE: para realizar borrados del conjunto de datos
- Existen otros tipos de comandos o métodos, como por ejemplo PATCH (similar a PUT, pero para cambios parciales), HEAD (para consultar sólo el encabezado de la respuesta obtenida), etc. Nos centraremos de momento en los cuatro métodos principales anteriores

Por lo tanto, identificando el recurso a solicitar y el comando a aplicarle, el servidor que ofrece esta API REST proporciona una respuesta a esa petición. Esta respuesta típicamente viene dada por un mensaje en formato JSON o XML (aunque éste cada vez está más en desuso).

Veremos cómo podemos identificar los diferentes tipos de comandos de nuestra API, y las URIs de los recursos a solicitar, para luego dar una respuesta en formato JSON ante cada petición.

Para simular peticiones de clientes, emplearemos una aplicación llamada Postman, que permite construir peticiones de diferentes tipos, empleando distintos tipos de comandos, cabeceras y contenidos, enviarlas al servidor que indiquemos y examinar la respuesta proporcionada por éste. En esta sesión veremos algunas nociones básicas sobre cómo utilizar Postman.

1.3. El formato JSON

JSON son las siglas de *JavaScript Object Notation*, una sintaxis propia de Javascript para poder representar objetos como cadenas de texto, y poder así serializar y enviar información de objetos a través de flujos de datos (archivos de texto, comunicaciones cliente-servidor, etc).

Un objeto Javascript se define mediante una serie de propiedades y valores. Por ejemplo, los datos de una persona (como nombre y edad) podríamos almacenarlos así:

```
let persona = {  
  nombre: "Nacho",  
  edad: 39  
};
```

Este mismo objeto, convertido a JSON, formaría una cadena de texto con este contenido:

```
{"nombre":"Nacho","edad":39}
```

Del mismo modo, si tenemos una colección (vector) de objetos como ésta:

```
let personas = [  
  { nombre: "Nacho", edad: 39 },  
  { nombre: "Mario", edad: 4 },  
  { nombre: "Laura", edad: 2 },  
  { nombre: "Nora", edad: 10 }  
];
```

```
]
```

Transformada a JSON sigue la misma sintaxis, pero entre corchetes:

```
[{"nombre":"Nacho","edad":39}, {"nombre":"Mario","edad":4},  
 {"nombre":"Laura","edad":2}, {"nombre":"Nora","edad":10}]
```

1.3.1. Conversión a y desde JSON

Javascript proporciona un par de métodos muy útiles para convertir desde JSON a objetos Javascript y viceversa. Veamos el ejemplo con el vector de personas anterior:

```
let personas = [  
  { nombre: "Nacho", edad: 39},  
  { nombre: "Mario", edad: 4},  
  { nombre: "Laura", edad: 2},  
  { nombre: "Nora", edad: 10}  
]
```

Si queremos convertir este vector en una cadena JSON, empleamos el método **JSON.stringify**. Recibe como parámetro un objeto Javascript, y devuelve la cadena JSON resultado. Si imprimimos por consola o pantalla esta cadena, veremos el formato JSON resultante (el mismo que se ha visto anteriormente):

```
let personasJSON = JSON.stringify(personas);  
console.log(personasJSON);
```

La operación inversa también es posible, mediante el método **JSON.parse**, que recibe como parámetro una cadena JSON y devuelve un objeto (o vector de objetos, según sea el caso) con los datos procesados. Por ejemplo, si creamos una nueva variable parseando la cadena JSON anterior, tendremos un objeto igual que el original:

```
let personas2 = JSON.parse(personasJSON);  
console.log(personas2);
```

En los siguientes ejemplos vamos a realizar comunicaciones cliente-servidor donde el cliente va a solicitar al servidor una serie de servicios, y éste responderá devolviendo un contenido en formato JSON. Sin embargo, gracias al framework Express que utilizaremos, la conversión desde un formato a otro será automática, y no tendremos que preocuparnos de utilizar estos métodos de conversión.

2. ¿Qué es Express?

Express es un framework ligero y, a la vez, flexible y potente para desarrollo de aplicaciones web con Node. En primer lugar, se trata de un framework ligero porque no viene cargado de serie con toda la funcionalidad que un framework podría tener, a diferencia de otros frameworks más pesados y autocontenidos como Symfony o Ruby on Rails. Pero, además, se trata de un framework flexible y potente porque permite añadirle, a través de módulos Node y de *middleware*, toda la funcionalidad que se requiera para cada tipo de aplicación. De este modo, podemos utilizarlo en su versión más ligera para aplicaciones web sencillas, y dotarle de más elementos para desarrollar aplicaciones muy complejas.

Como veremos, con Express podemos desarrollar tanto servidores típicos de contenido estático (HTML, CSS y Javascript), como servicios web accesibles desde un cliente, y por supuesto, aplicaciones que combinen ambas cosas.

Podéis encontrar información actualizada sobre Express, tutoriales y demás información en su [página oficial](#).

2.1. Descarga e instalación

La instalación de express es tan sencilla como la de cualquier otro módulo que queramos incorporar a un proyecto Node. Simplemente necesitamos ejecutar el correspondiente comando `npm install` (y, previamente, el comando `npm init` en el caso de que aún no hayamos creado el archivo "package.json"):

```
npm install express
```

2.1.1. Ejemplo de servidor básico con Express

Crea un proyecto llamado "PruebaExpress" en tu carpeta de "ProyectosNode/Pruebas", e instala Express en él. Después, crea un archivo llamado "index.js" y deja en él este código:

```
const express = require('express');  
  
let app = express();  
  
app.listen(8080);
```

El código, como ves, es muy sencillo. Primero incluimos la librería, después inicializamos una instancia de Express (normalmente se almacena en una variable llamada `app`), y finalmente, la ponemos a escuchar por el puerto que queramos. En este caso, se ha escogido el puerto 8080 para no interferir con el puerto por defecto por el que se escuchan las peticiones HTTP, que es el 80. También es habitual encontrar ejemplos de código en Internet que usan los puertos 3000 o 6000 para las pruebas. Es indiferente el número de puerto, siempre que no interfiera con otro servicio que tengamos en marcha en el sistema.

Para probar el ejemplo desde nuestra máquina local, basta con abrir un navegador y acceder a la URL:

`http://localhost:8080`

Si pruebas a ejecutar la aplicación, verás que no finaliza. Hemos creado un pequeño servidor Express que queda a la espera de peticiones de los clientes. Sin embargo, aún no está preparado para responder a ninguna de ellas. Eso lo haremos en los siguientes apartados.

2.2. Express como proveedor de servicios

Ahora que ya sabemos qué es Express y cómo incluirlo en las aplicaciones Node, veremos uno de los principales usos que se le da: el de servidor que proporciona servicios REST a los clientes que lo soliciten.

Para ello, y como paso previo, debemos comprender y asimilar cómo se procesan las rutas en Express, y cómo se aísla el tratamiento de cada una, de forma que el código resulta muy modular e independiente entre rutas.

Recordemos, antes de nada, la estructura básica que tiene un servidor Express:

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

let app = express();

app.listen(8080);
```

2.2.1. Un primer servicio básico

Partiendo de la base anterior, vamos a añadir una serie de rutas en nuestro servidor principal para dar soporte a los servicios asociados a las mismas. Una vez hemos inicializado la aplicación (variable `app`), basta con ir añadiendo métodos (`get`, `post`, `put` o `delete`), indicando para cada uno la ruta que debe atender, y el callback o función que se ejecutará en ese caso. Por ejemplo, para atender a una ruta llamada `/bienvenida` por GET, añadiríamos este método:

```
let app = express();

app.get('/bienvenida', (req, res) => {
  res.send('Hola, bienvenido/a');
});

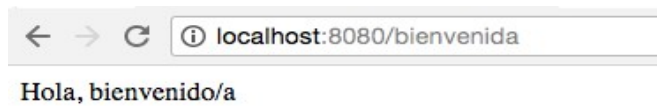
...

```

El callback en cuestión recibe dos parámetros siempre: el objeto que contiene la petición (típicamente llamado `req`, abreviatura de *request*), y el objeto para emitir la respuesta (típicamente llamado `res`, abreviatura de *response*). Más adelante veremos qué otras cosas podemos hacer con estos objetos, pero de momento emplearemos la respuesta para enviar (`send`) texto al cliente que solicitó el servicio, y `req` para obtener determinados datos de la petición.

Podemos volver a lanzar el servidor Express, y probar este nuevo servicio accediendo a la URL correspondiente:

`http://localhost:8080/bienvenida`



Del mismo modo, se añadirán el resto de métodos para atender las distintas opciones de la aplicación. Por ejemplo:

```
app.delete('/comentarios', (req, res) => { ...
```

En este punto, puedes realizar el [Ejercicio 1](#) de los propuestos al final de la sesión.

2.3. Elementos básicos: aplicación, petición y respuesta

Existen tres elementos básicos sobre los que se sustenta el desarrollo de aplicaciones en Express: la aplicación en sí, el objeto con la petición del cliente, y el objeto con la respuesta a enviar.

2.3.1. La aplicación

La aplicación es una instancia de un objeto Express, que típicamente se asocia a una variable llamada `app` en el código:

```
const express = require('express');  
let app = express();
```

Toda la funcionalidad de la aplicación (métodos de respuesta a peticiones, inclusión de *middleware*, etc) se asienta sobre este elemento. Cuenta con una serie de métodos útiles, que iremos viendo en futuros ejemplos, como son:

- `use(middleware)`: para incorporar *middleware* al proyecto
- `set(propiedad, valor) / get(propiedad)`: para establecer y obtener determinadas propiedades relativas al proyecto
- `listen(puerto)`: para hacer que el servidor se quede escuchando por un puerto determinado.
- `render(vista, [opciones], callback)`: para mostrar una determinada vista estática como respuesta, pudiendo especificar opciones adicionales y un callback de respuesta
- ...

2.3.2. La petición

El objeto de petición (típicamente lo encontraremos en el código como `req`) se crea cuando un cliente envía una petición a un servidor Express. Contiene varios métodos y propiedades útiles para acceder a información contenida en la petición, como:

- `params`: la colección de parámetros que se envía con la petición
- `query`: con la *query string* enviada en una petición GET
- `body`: con el cuerpo enviado en una petición POST
- `files`: con los archivos subidos desde un formulario en el cliente

- `get(cabecera)`: un método para obtener distintas cabeceras de la petición, a partir de su nombre
- `path`: para obtener la ruta o URI de la petición
- `url`: para obtener la URI junto con cualquier *query string* que haya a continuación
- ...

2.3.3. La respuesta

El objeto respuesta se crea junto con el de la petición, y se completa desde el código del servidor Express con la información que se vaya a enviar al cliente. Típicamente se representa con la variable `res`, y cuenta, entre otros, con estos métodos y propiedades de utilidad:

- `status(codigo)`: establece el código de estado de la respuesta
- `set(cabecera, valor)`: establece cada una de las cabeceras de respuesta que se necesiten
- `redirect (estado, url)`: redirige a otra URL, con el correspondiente código de estado
- `send([estado], cuerpo)`: envía el contenido indicado, junto con el código de estado asociado (de forma opcional, si no se envía éste por separado).
- `json([estado], cuerpo)`: envía contenido JSON específicamente, junto con el código de estado asociado (opcional)
- ...

3. Ejemplo de enrutamiento simple

En este apartado veremos cómo emplear un enrutamiento simple para ofrecer diferentes servicios empleando Mongoose contra una base de datos MongoDB. Para ello, crearemos una carpeta llamada "PruebaContactosExpress" en nuestra carpeta de pruebas ("ProyectosNode/Pruebas"). Instalaremos Express y Mongoose en ella, lo que puede hacerse con un simple comando (aunque previamente necesitaremos haber ejecutado `npm init` para crear el archivo "package.json"):

```
npm install mongoose express
```

Nuestra aplicación tendrá un archivo "index.js" donde incorporaremos tanto el modelo de datos de contactos (el mismo visto en sesiones previas) como las rutas para responder a diferentes operaciones sobre los contactos.

NOTA: en posteriores sesiones veremos que existe otra forma más adecuada de estructurar una aplicación Node/Express, especialmente cuando el número de colecciones y/o operaciones sobre las mismas es grande. Pero de momento vamos a optar por añadirlo todo a un mismo archivo para simplificar el código en estos pasos iniciales

3.1. Esqueleto básico del servidor principal

Vamos a definir la estructura básica que va a tener nuestro servidor "index.js", antes de añadirle las rutas para dar respuesta a los servicios. Esta estructura consistirá en incorporar Express y Mongoose, conectar con la base de datos y definir el modelo y esquema para los contactos:

```
const express = require('express');
const mongoose = require('mongoose');

mongoose.Promise = global.Promise;
mongoose.connect('mongodb://localhost:27017/contactos');

const mongoose = require('mongoose');

let contactoSchema = new mongoose.Schema({
  nombre: {
    type: String,
    required: true,
    unique: true,
    minlength: 1,
    trim: true
  },
  telefono: {
    type: String,
    required: true,
    trim: true,
    match: /^\\d{9}$/
  },
  edad: {
    type: Number,
```

```

        min: 18,
        max: 120
    }
});

let Contacto = mongoose.model('contacto', contactoSchema);

let app = express();

app.listen(8080);

```

3.1.1. Servicios a desarrollar

A continuación, definiremos los servicios a los que responderá nuestro servidor. Ofreceremos los siguientes servicios, para dar cabida a todas las operaciones básicas que podríamos hacer con los contactos

- Listado de todos los contactos
- Listado de un contacto a partir de su *id*
- Inserción de un nuevo contacto
- Modificación de datos de un contacto
- Borrado de un contacto

3.2. Servicios de listado (GET)

3.2.1. Listado de todos los contactos

El servicio que lista todos los contactos es el más sencillo: atenderemos por GET a la URI */contactos*, y en el código haremos un *find* de todos los contactos, devolviéndolo directamente como resultado:

```

app.get('/contactos', (req, res) => {
    Contacto.find().then(resultado => {
        res.send(resultado);
    }).catch (error => {
        res.send([]);
    });
});

```

Observad que enviamos como respuesta el array de objetos obtenido, y es Express el que se encarga de transformarlo a JSON por nosotros. En caso de error, enviaremos un array vacío.

3.2.2. Ficha de un contacto a partir de su *id*

Veamos ahora cómo procesar con Express URIs dinámicas. En este caso, accederemos por GET a una URI con el formato */contactos/:id*, siendo *:id* el *id* del contacto que queremos obtener. Si especificamos la URI con ese mismo formato en Express, automáticamente se le asocia al parámetro que venga a continuación de */contactos* el nombre *id*, con lo que podemos acceder a él directamente por el objeto **req.params** de la petición. De este modo, el servicio queda así de simple:

```

app.get('/contactos/:id', (req, res) => {
    Contacto.findById(req.params.id).then(resultado => {

```

```

    if(resultado)
      res.send({error: false, resultado: resultado});
    else
      res.send({error: true,
        mensajeError: "No se han encontrado contactos"});
  }).catch (error => {
    res.send({error: true,
      mensajeError: "Error buscando el contacto indicado"});
  });
});

```

Notar que lo que devolvemos en la respuesta es un objeto Javascript con tres campos:

- `error`, que establece si ha habido algún error procesando la petición (`true`) o no (`false`)
- `mensajeError`, que está presente sólo si `error` es `true`. En ese caso, almacena el mensaje de error concreto que se ha producido.
- `resultado`, que está presente si `error` es `false`. Contiene el resultado de la búsqueda.

Si la llamada a `findById` funciona correctamente, iremos a la sección `then`, y comprobaremos si hay un resultado válido. Si lo hay, establecemos `error=false` y en caso contrario haremos `error=true` con el correspondiente mensaje de error. Si algo falla y se lanza una excepción, iremos a la cláusula `catch` y enviaremos el correspondiente mensaje de error al cliente.

3.2.3. Uso de la query string para pasar parámetros

En el caso de querer pasar los parámetros en la *query string* (es decir, por ejemplo, `/contactos?id=XXX`) no hay forma de establecer dichos parámetros en la URI del método `get`. En ese caso deberemos comprobar si existe el parámetro correspondiente dentro del objeto `req.query`:

```

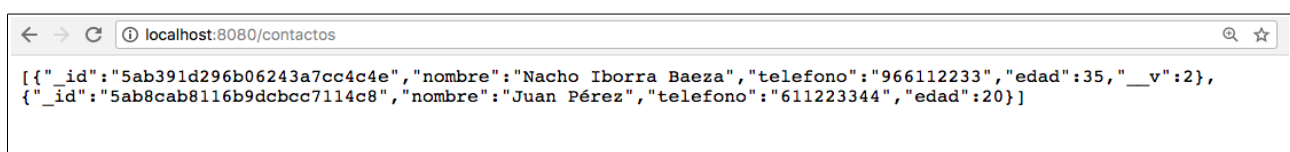
app.get('/contactos', (req, res) => {
  if(req.query.id) {
    // Buscar por id
  }
  else {
    // Listado general de contactos
  }
});

```

3.2.4. Prueba de los servicios

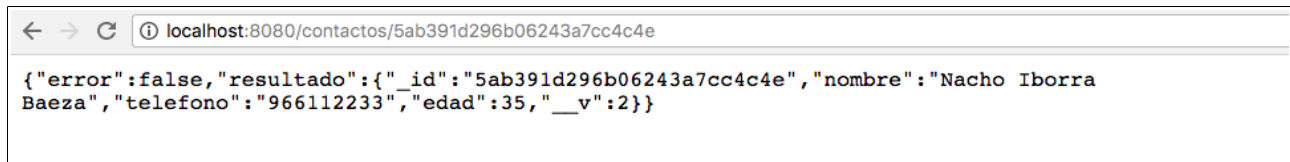
Estos dos servicios de listado (general y por *id*) se pueden probar fácilmente desde un navegador web. Basta con poner en marcha el servidor Node, abrir un navegador y acceder a esta URL para el listado general:

`http://localhost:8080/contactos`



O a esta otra para la ficha de un contacto (sustituyendo el *id* en negrita de la URL por uno correcto que exista en la base de datos):

<http://localhost:8080/contactos/5ab391d296b06243a7cc4c4e>



En este último caso, observa que:

- Si pasamos un *id* que no exista, nos indicará con un mensaje de error que "No se han encontrado contactos"
- Si pasamos un *id* que no sea adecuado (por ejemplo, que no tenga 12 bytes), obtendremos una excepción, y por tanto el mensaje de "Error buscando el contacto indicado".

3.3. Servicios de inserción, borrado y modificación

3.3.1. Inserción de contactos (POST)

Vamos a insertar un nuevo contacto, pasando en el cuerpo de la petición los datos del mismo (nombre, teléfono y edad) en formato JSON. En esta ocasión, vamos a valernos de un *middleware* para Express llamado *body-parser*, que facilita el procesamiento del cuerpo de las peticiones para acceder directamente a los datos que se envían en ella. En primer lugar, deberemos instalar el módulo en nuestro proyecto:

```
npm install body-parser
```

Después, lo incluimos con `require` junto al resto:

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
...
```

Y finalmente, lo añadimos como *middleware* con `app.use`, justo después de inicializar la app. Como lo que vamos a hacer es trabajar con objetos JSON, añadiremos el procesador JSON de *body-parser*:

```
let app = express();
app.use(bodyParser.json());
...
```

Ahora vamos a nuestro servicio POST. Observa qué sencillo queda empleando este *middleware*:

```
app.post('/contactos', (req, res) => {
  let nuevoContacto = new Contacto({
    nombre: req.body.nombre,
    telefono: req.body.telefono,
    edad: req.body.edad
  });
  nuevoContacto.save().then(resultado => {
    res.send({error: false, resultado: resultado});
  }).catch(error => {
    res.send({error: true,
```

```

        mensajeError: "Error añadiendo contacto"}));
    });
});

```

Al principio, construimos el contacto a partir de los datos JSON que llegan en el cuerpo, accediendo a cada campo por separado con `req.body.nombre_campo`, gracias a *body-parser*. El resto del código es el que ya conoces de ejemplos previos con Mongoose (llamada a `save` y procesamiento de la promesa)

3.3.2. Modificación de contactos (PUT)

La modificación de contactos es estructuralmente muy similar a la inserción: enviaremos en el cuerpo de la petición los datos nuevos del contacto a modificar (a partir de su *id*), y utilizaremos "body-parser" para obtenerlos, y llamar a los métodos apropiados de Mongoose para realizar la modificación del contacto. La URI a la que asociaremos este servicio será similar a la del POST, pero añadiendo el *id* del contacto que queramos modificar:

```

app.put('/contactos/:id', (req, res) => {
  Contacto.findByIdAndUpdate(req.params.id, {
    $set: {
      nombre: req.body.nombre,
      telefono: req.body.telefono,
      edad: req.body.edad
    }
  }, {new: true}).then(resultado => {
    res.send({error: false, resultado: resultado});
  }).catch(error => {
    res.send({error: true,
      mensajeError: "Error actualizando contacto"});
  });
});

```

Como se puede ver, obtenemos el *id* desde los parámetros (`req.params`) como cuando consultábamos la ficha de un contacto, y utilizamos dicho *id* para buscar al contacto en cuestión y actualizar sus campos con `findByIdAndUpdate`. En este punto, volvemos a hacer uso de la librería "body-parser" para procesar los datos que llegan desde el cuerpo de la petición. Tras la llamada al método, devolvemos un objeto JSON con los tres atributos que ya hemos visto en servicios anteriores (`error`, `mensajeError` y `resultado`).

3.3.3. Más sobre "body-parser"

Existen otras posibilidades de uso del *middleware body-parser*. En los ejemplos anteriores lo hemos empleado para procesar cuerpos con formato JSON. Pero es posible también que empleemos formularios tradicionales HTML, que envían los datos como si fueran parte de una *query-string*, pero por POST:

nombre=Nacho&telefono=911223344&edad=39

Para procesar contenidos de este otro tipo, basta con añadir de nuevo la librería como *middleware*, indicando en este caso otro método:

```

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: false}));

```

El parámetro *extended* indica qué tipo de *parser* queremos utilizar para procesar los datos de la petición: si lo dejamos a *false*, emplearemos la librería *querystring*, y si está a *true*, se empleará la librería *qs*, con algunas opciones algo más avanzadas para incluir objetos más complejos.

En cualquier caso, deberemos asegurarnos de que el tipo de contenido de la petición se ajusta al *middleware* correspondiente: para peticiones en formato JSON, el contenido deberá ser *application/json*, mientras que para enviar los datos del formulario en formato *query-string*, el tipo deberá ser *application/x-www-form-urlencoded*. Si añadimos los dos *middlewares* (tanto para JSON como para *urlencoded*), entonces se activará uno u otro automáticamente, dependiendo del tipo de petición que llegue desde el cliente.

3.3.4. Borrado de contactos (DELETE)

Para el borrado de contactos, emplearemos una URI similar a la ficha de un contacto o a la actualización, pero en este caso asociada al comando DELETE. Le pasaremos el *id* del contacto a borrar. En este caso, obtendremos el *id* también de *req.params*, y buscaremos y eliminaremos el contacto indicado.

```
app.delete('/contactos/:id', (req, res) => {
  Contacto.findByIdAndRemove(req.params.id).then(resultado => {
    res.send({error: false, resultado: resultado});
  }).catch(error => {
    res.send({error: true,
      mensajeError: "Error eliminando contacto"});
  });
});
```

En este punto puedes realizar el [Ejercicio 2](#) de los propuestos al final de la sesión, aunque no podrás probar su funcionamiento (salvo para los servicios GET) hasta que completes el siguiente apartado, que trata sobre la aplicación Postman para probar servicios.

3.3.5. Sobre el resultado de la actualización o el borrado

En los ejemplos anteriores para PUT y DELETE, tras la llamada a *findByIdAndUpdate* o *findByIdAndRemove*, nos hemos limitado a devolver el resultado en la cláusula *then*. Sin embargo, conviene tener en cuenta que, si proporcionamos un *id* válido pero que no exista en la base de datos, el código de esta cláusula también se ejecutará, pero el objeto *resultado* será nulo (*null*). Podemos, por tanto, diferenciar con un *if..else* si el resultado es correcto o no, y mostrar una u otra cosa:

```
if (resultado)
  res.send({error: false, resultado: resultado});
else
  res.send({error: true,
    mensajeError: "No se ha encontrado el contacto"});
```

4. Probar los servicios con Postman

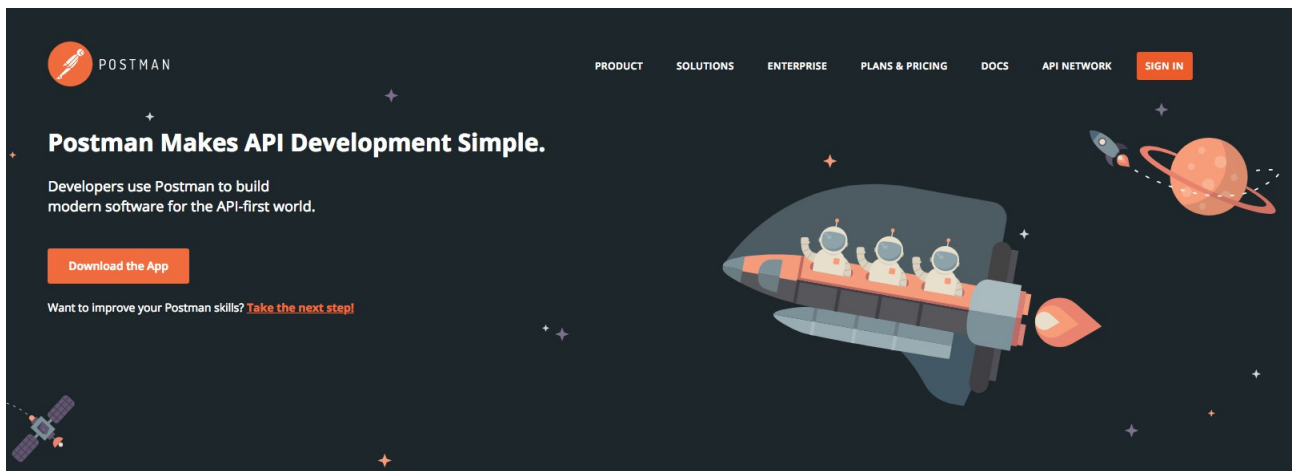
Ya hemos visto que probar unos servicios de listado (GET) es sencillo a través de un navegador. Incluso probar un servicio de inserción (POST) podría hacerse a través de un formulario HTML, pero los servicios de modificación (PUT) o borrado (DELETE) exigen de otras herramientas para poder ser probados. Una de las más útiles en este sentido es Postman.



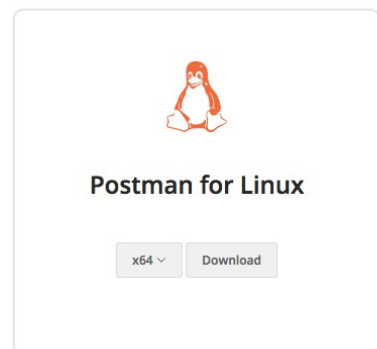
Postman es una aplicación gratuita y multiplataforma que permite enviar todo tipo de peticiones de clientes a un servidor determinado, y examinar la respuesta que éste produce. De esta forma, podemos comprobar que los servicios ofrecen la información adecuada antes de ser usados por una aplicación cliente real.

4.1. Descarga, instalación y primeros pasos

Para descargar e instalar Postman, debemos ir a su [web oficial](#), y hacer clic en el botón de descargar (*Download the app*), y después elegir la versión concreta para nuestro sistema operativo.

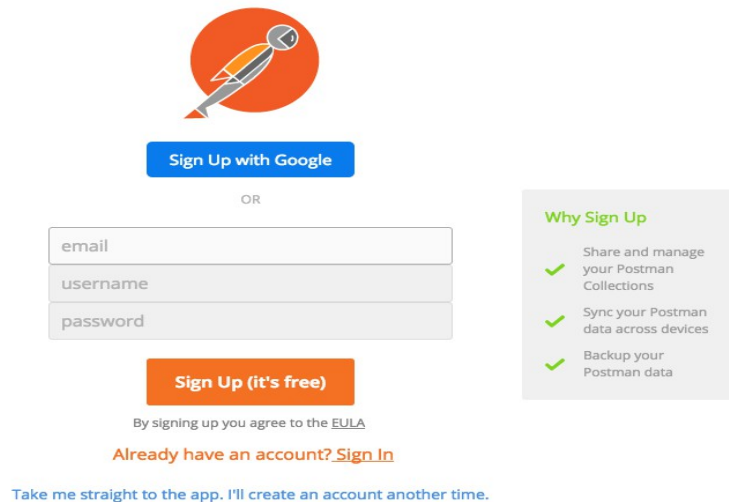


Choose your platform:



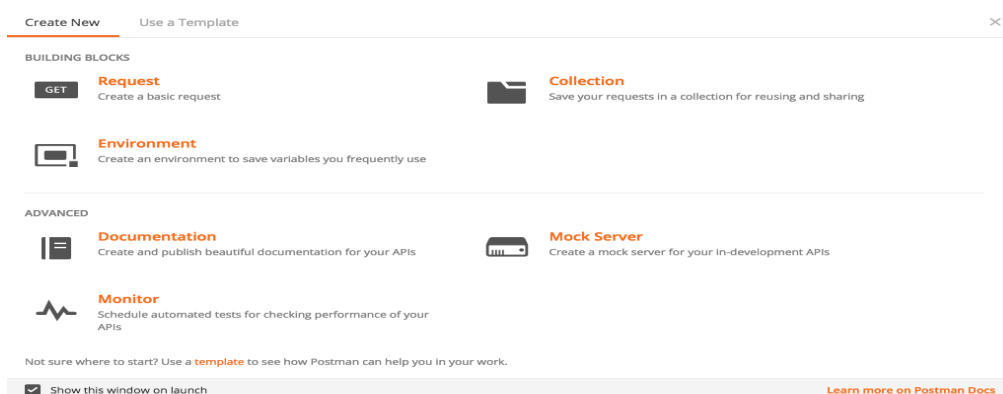
Obtendremos un archivo comprimido portable, que podemos descomprimir y ejecutar directamente. Nada más iniciar, nos preguntará si queremos asociar Postman a una

cuenta Google. Esto tiene las ventajas de poder almacenar en la cuenta las diferentes pruebas que hagamos, para luego poderlas utilizar en otros equipos, pero no es un paso obligatorio, y podemos omitirlo si queremos pulsando el enlace azul inferior

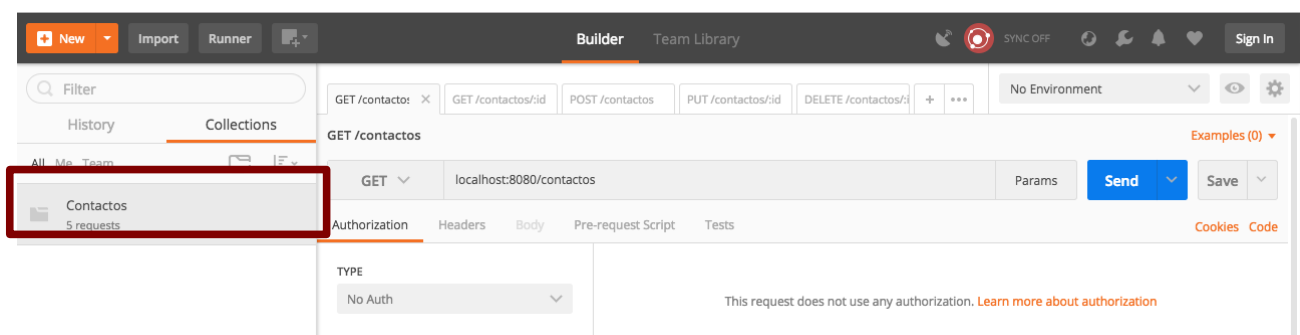


The image shows the Postman sign-up interface. At the top is the Postman logo (an orange circle with a white key icon). Below it is a blue button labeled "Sign Up with Google". Underneath is the word "OR". There are three input fields for "email", "username", and "password". Below these is an orange button labeled "Sign Up (it's free)". Underneath that is the text "By signing up you agree to the [EULA](#)". Below that is a link "Already have an account? [Sign In](#)". At the bottom is a link "Take me straight to the app. I'll create an account another time." To the right of the sign-up form is a box titled "Why Sign Up" with three green checkmarks and text: "Share and manage your Postman Collections", "Sync your Postman data across devices", and "Backup your Postman data".

Tras iniciar la aplicación, veremos un diálogo para crear peticiones simples o colecciones de peticiones (conjuntos de pruebas para una aplicación). Lo que haremos habitualmente será esto último.

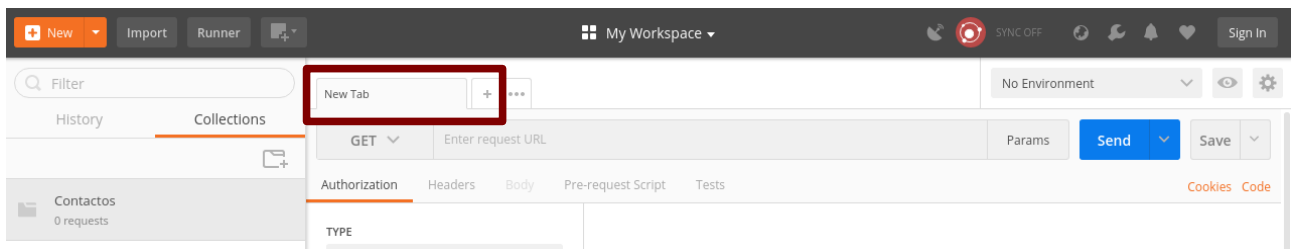


Si elegimos crear una colección, le deberemos asociar un nombre (por ejemplo, "Contactos", y guardarla. Entonces podremos ver la colección en el panel izquierdo de Postman:



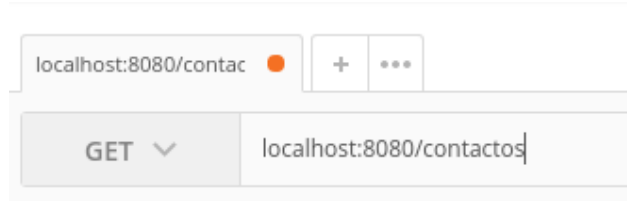
Desde el botón "New" en la esquina superior izquierda podemos crear nuevas peticiones (también nuevas colecciones) y asociarlas a una colección. Existe una forma alternativa

(quizá más cómoda) de crear esas peticiones, a través del panel de pestañas, añadiendo nuevas:

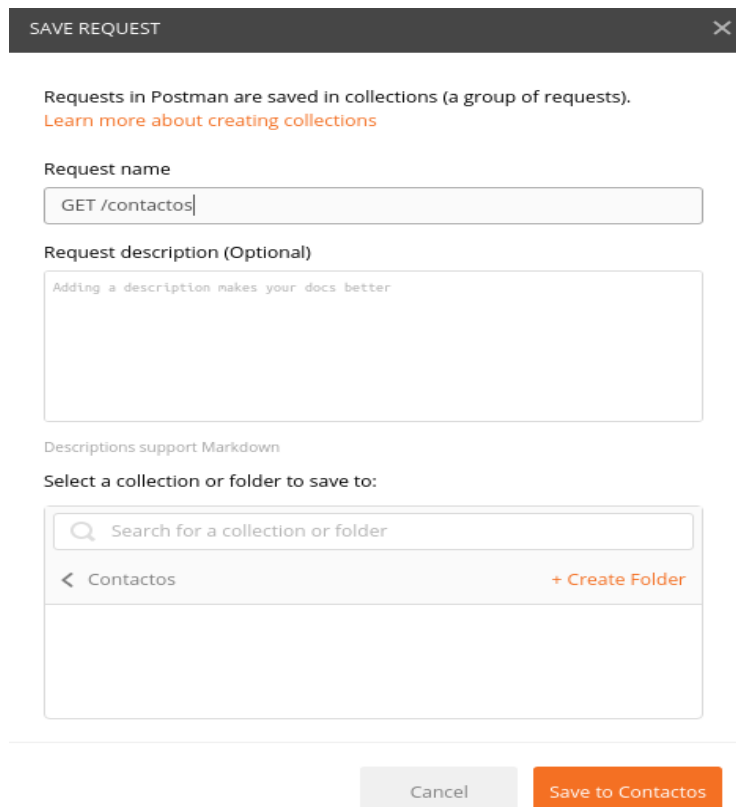


4.2. Añadir peticiones simples: GET

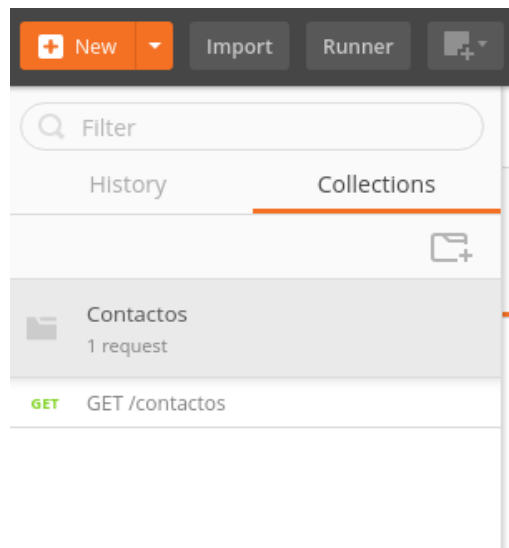
Para añadir una petición, habitualmente elegiremos el tipo de comando bajo las pestañas (GET, POST, PUT, DELETE) y la URL asociada a dicho comando. Por ejemplo:



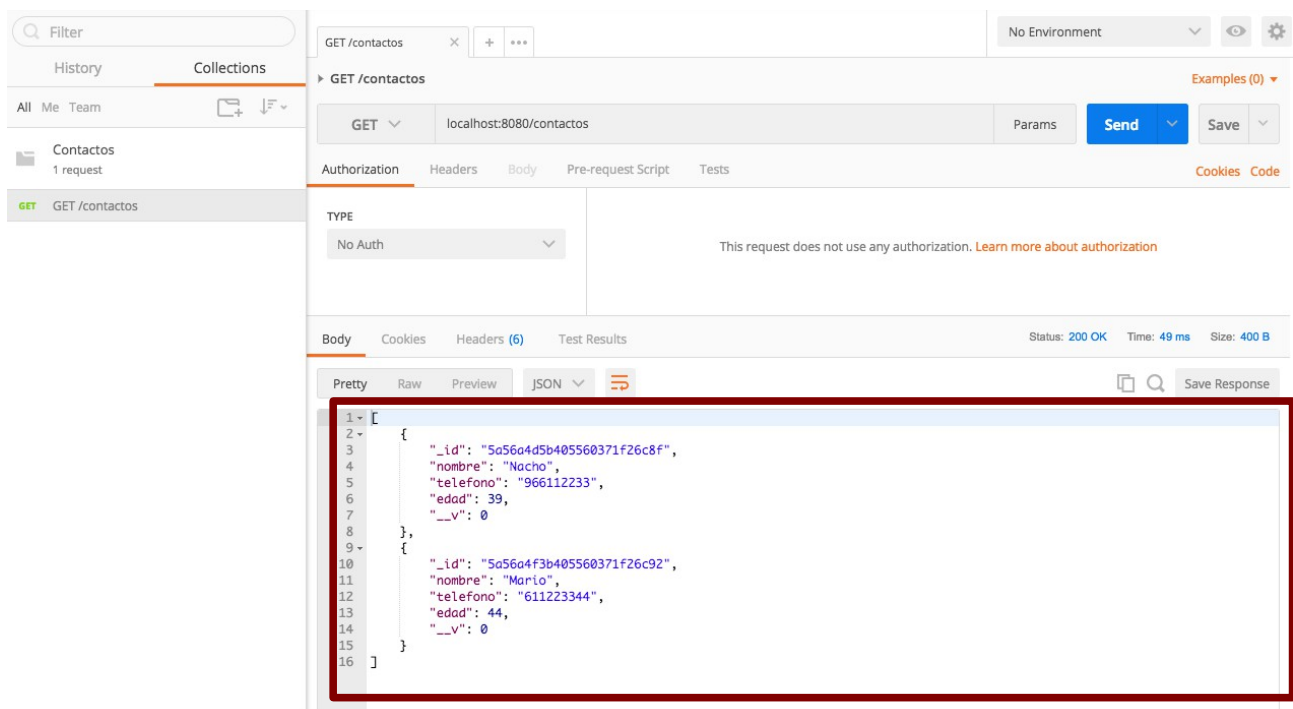
Entonces, podemos hacer clic en el botón "Save" en la parte derecha, y guardar la petición para poderla reutilizar. Al guardarla, nos pedirá que le asignemos un nombre (por ejemplo, "GET /contactos" en este caso), y la colección en la que se almacenará (nuestra colección de "Contactos").



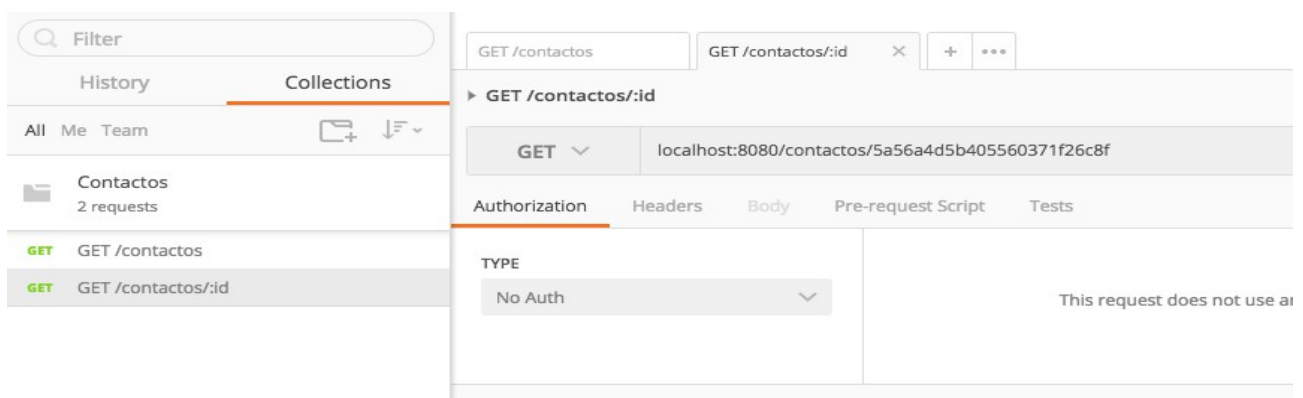
Después, podremos ver la prueba asociada a la colección, en el panel izquierdo:



Si seleccionamos esta prueba y pulsamos en el botón azul de "Send" (parte derecha), podemos ver la respuesta emitida por el servidor en el panel inferior de respuesta:



Siguiendo estos mismos pasos, podemos también crear una nueva petición para obtener un contacto a partir de su *id*, por GET:



Bastaría con reemplazar el *id* de la URL por el que queramos consultar realmente. Si probamos esta petición, obtendremos la respuesta correspondiente:

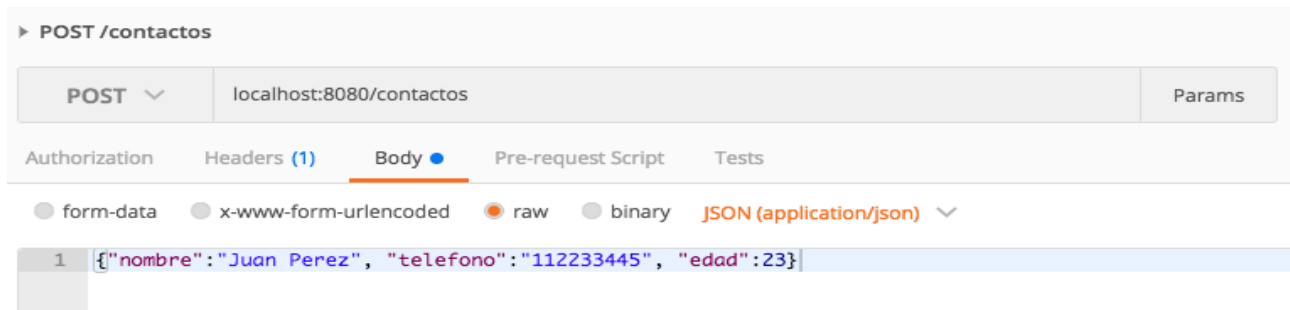
```
1 {
2   "error": false,
3   "resultado": {
4     "_id": "5a56a4d5b405560371f26c8f",
5     "nombre": "Nacho",
6     "telefono": "966112233",
7     "edad": 39,
8     "__v": 0
9   }
10 }
```

4.3. Añadir peticiones POST

Las peticiones POST difieren de las peticiones GET en que se envía cierta información en el cuerpo de la petición. Esta información normalmente son los datos que se quieren añadir en el servidor. ¿Cómo podemos hacer esto con Postman?

En primer lugar, creamos una nueva petición, elegimos el comando POST y definimos la URL (en este caso, *localhost:8080/contactos*). Entonces, hacemos clic en la pestaña *Body*, bajo la URL, y establecemos el tipo como *raw* para que nos deje escribirlo sin restricciones. También conviene cambiar la propiedad *Text* para que sea *application/json*, y que así el servidor recoja el tipo de dato adecuado y se active el "body-parser" correspondiente. Se añadirá automáticamente una cabecera de petición (*Header*) que especificará que el tipo de contenido que se va a enviar son datos JSON.

Después, en el cuadro de texto bajo estas opciones, especificamos el objeto JSON que queremos enviar para insertar:



Si enviamos esta petición, obtendremos el resultado de la inserción:

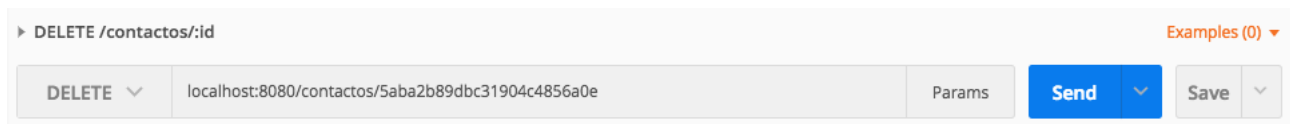
```
1 {
2   "error": false,
3   "resultado": {
4     "_id": "5aba2b89dbc31904c4856a0e",
5     "nombre": "Juan Perez",
6     "telefono": "112233445",
7     "edad": 23,
8     "__v": 0
9   }
10 }
```

4.4. Añadir peticiones PUT o DELETE

En el caso de peticiones PUT, procederemos de forma similar a las peticiones POST vistas antes: debemos elegir el comando (PUT en este caso), la URL, y completar el cuerpo de la petición con los datos que queramos modificar del contacto. En este caso, además, el *id* del contacto lo enviaremos también en la propia URL:

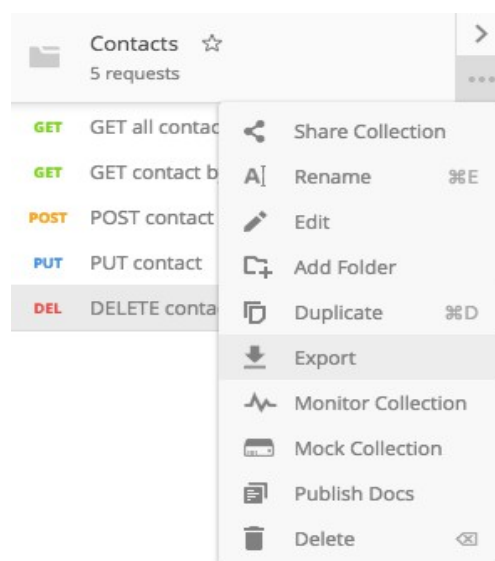


Para peticiones DELETE, la mecánica es similar a la ficha del contacto, cambiando el comando GET por DELETE, y sin necesidad de establecer nada en el cuerpo de la petición:

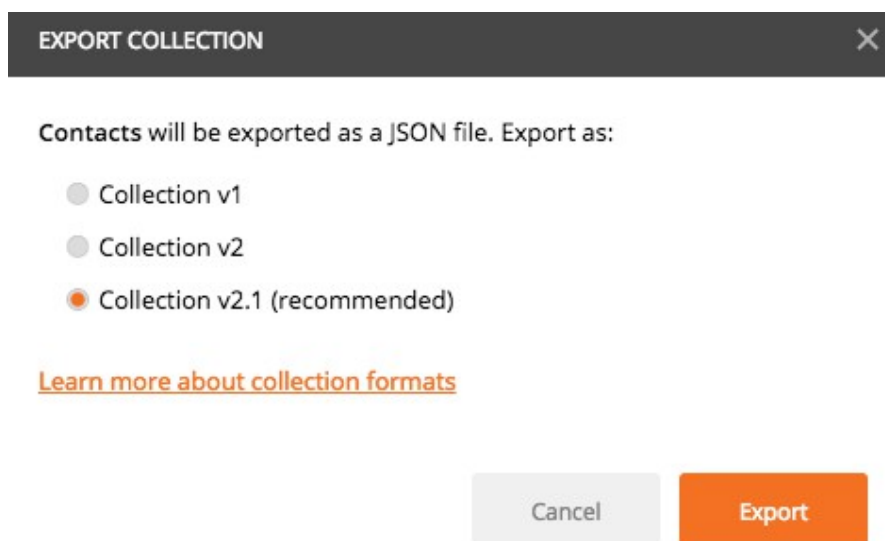


4.5. Exportar/Importar colecciones

Podemos exportar e importar nuestras colecciones en Postman, de forma que podemos llevarlas de un equipo a otro. Para **exportar** una colección, hacemos clic en el botón de puntos suspensivos (...) que hay junto a ella en el panel izquierdo, y elegimos *Export*.

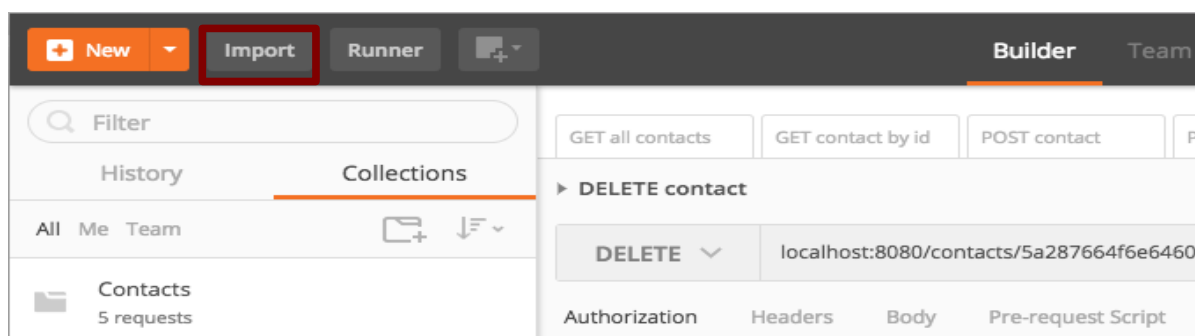


Nos preguntará para qué versión de Postman queremos exportar (normalmente la recomendada es la mejor opción):



Se creará un nuevo archivo Postman en la ubicación que elijamos.

Si queremos **importar** una colección previamente exportada, podemos hacer clic en el botón *Import* de la esquina superior izquierda en la ventana principal:



Entonces, elegimos el archivo Postman con la colección y aparecerá en el panel izquierdo tras la importación.

En este punto, puedes realizar el [Ejercicio 3](#) de los propuestos al final de la sesión.

5. Ejercicios

Para los ejercicios de esta sesión, crearemos una carpeta llamada "Sesion4" en nuestra carpeta de "ProyectosNode/Ejercicios", y dentro ubicaremos los distintos proyectos que se nos indique.

5.1. Ejercicio 1

Crea una carpeta llamada "Ejercicio_4_1" en la carpeta de ejercicios "ProyectosNode/Ejercicios/Sesion4". Instala Express en ella, y define un servidor básico que responda por GET a estas dos URIs:

- URI `/fecha`: el servidor enviará como respuesta al cliente la fecha y hora actuales. Puedes utilizar el tipo `Date` de Javascript Standard sin más, o también puedes "recrearte" con la librería "moment" vista en sesiones anteriores, si quieres.
- URI `/usuario`: el servidor enviará el login del usuario que entró al sistema. Necesitarás emplear la librería "os" del núcleo de Node, vista en sesiones anteriores, para obtener dicho usuario.

5.2. Ejercicio 2

Crea una carpeta llamada "Ejercicio_4_2" en la carpeta de ejercicios "ProyectosNode/Ejercicios/Sesion4". Instala Express y Mongoose en dicha carpeta, y define un servidor Express que proporcione los siguientes servicios sobre la base de datos de libros creada en la sesión anterior:

- Listar todos los libros. Accederá por GET a la URI `/libros`
- Buscar un libro por su *id*. Accederá por GET a la URI `/libros/:id`
- Insertar un nuevo libro. Accederá por POST a la URI `/libros`
- Modificar un libro a partir de su *id*. Accederá por PUT a la URI `/libros/:id`
- Borrar un libro a partir de su *id*. Accederá por DELETE a la URI `/libros/:id`

Observa que los servicios son los mismos que los realizados para el ejemplo de contactos de esta sesión, pero empleando la base de datos de libros.

5.3. Ejercicio 3

Crea una nueva colección en Postman llamada "Libros", y define las siguientes peticiones:

- GET `/libros`
- GET `/libros:id`
- POST `/libros`
- PUT `/libros/:id`
- DELETE `/libros/:id`

Comprueba que el funcionamiento de todas estas pruebas es el correcto. Una vez funcione todo, exporta la colección a un archivo, que será lo que deberás entregar para este ejercicio.