3. Desarrollo de servicios con Node.js

Parte III. Utilizar Express para implementar servicios

Programación de Servicios y Procesos

Nacho Iborra
Álvaro Pérez
Javier Carrasco

Tabla de contenido

8. Introducción a los servicios REST

En esta última sesión de la unidad se aprenderá cómo usar ***Express*** para construir un servidor web Node que usa MongoDB para almacenar los datos de una aplicación, y Mongoose para acceder a estos datos, y brindar algunos servicios REST básicos a las aplicaciones cliente que los soliciten.

En sesiones anteriores, se vió cómo usar Node como una aplicación local, y se ha "interactuado" con esta aplicación a través de la línea de comandos, pasando algunos parámetros adicionales para pedirle que añada/enumere/elimine/actualice documentos… pero no es así como se utiliza Node en el mundo real.

Lo que suele hacerse con Node es crear servidores web, es decir, aplicaciones que se ejecutan en una máquina y escuchar a través de un puerto determinado, de modo que cualquier aplicación que se conecte a ese puerto y solicite algo, obtendrá una respuesta del servidor. Normalmente, estas aplicaciones se conectan a una URL determinada solicitando algún documento o recurso, y qué se hará aquí es crear una aplicación Node que obtenga estas solicitudes y genere la respuesta correspondiente.

8.1. Lo básico: HTTP y URL

Quizá algunos de vosotros aún no se hayan enfrentado a aplicaciones web. Si este es tu caso, debes saber que cada aplicación web se basa en una arquitectura cliente-servidor, en la que un servidor está esperando que los clientes se conecten a él, y los clientes se conectan a los servidores para solicitar algunos recursos.

Estas comunicaciones se realizan mediante un protocolo llamado HTTP, o HTTPS para comunicaciones seguras (encriptadas). En ambos casos, los clientes y los servidores envían información estándar en cada mensaje:

- En cuanto a los clientes, envían al servidor el recurso que desean obtener, junto con algunos datos adicionales, como encabezados de solicitud (con información sobre el tipo de cliente, contenido aceptado, etc), o parámetros adicionales llamados datos de formulario.
- En cuanto a los servidores, reciben estas solicitudes y devuelven información relevante como un código de estado (indicando si la solicitud ha sido procesada con éxito o no), encabezados de respuesta (con información sobre el tipo de contenido proporcionado, tamaño, idioma, etc) y el contenido en sí.

Para solicitar estos recursos, los clientes se conectan a una URL determinada (ubicación uniforme de recursos) en el servidor. Esta URL consiste en un fragmento de texto con tres secciones diferentes:

- El protocolo utilizado (http o https).
- El nombre de dominio, que identifica dónde se encuentra el servidor.

- La ruta de recursos, también denominada URI (*Uniform Resource Identifier*), que identifica el recurso concreto que desea el cliente, entre todos los recursos proporcionados por este servidor.

Por ejemplo, en la siguiente URL:

```
http://myserver.com/books?id=123
```

El protocolo es *http* y el nombre de dominio es *myserver.com*. Entonces, la ruta del recurso o URI es *books?id=123*, y la sección después del símbolo '?' es la información adicional llamada datos de formulario que se envía al servidor para especificar el recurso concreto que se está buscando. En este caso, se podría estar buscando un libro de un catálogo, cuyo identificador es 123.

8.2. Introducción a los servicios REST

Los servicios web son un tipo específico de recurso del que dependen muchas aplicaciones web. REST significa transferencia de estado representacional (*REpresentational State Transfer*) y se refiere a una arquitectura de aplicación distribuida basada en el protocolo HTTP. En este tipo de aplicaciones, se identifica cada recurso con una URI asociada, y se establece un conjunto limitado de comandos o métodos que se pueden enviar al servidor junto con esa URI. Los comandos/métodos más típicos son:

- ***GET***: para recuperar resultados, como listas o detalles sobre un elemento dado. Es equivalente a una instrucción *find* o *SELECT* en una base de datos.
- ***POST***: Para añadir elementos al servidor. Por lo general, corresponde a una instrucción *INSERT* en una base de datos.
- ***PUT***: para actualizar recursos en el servidor. Por lo general, está asociado con el comando *UPDATE* en una base de datos.
- ***DELETE***: para eliminar recursos del servidor, lo que generalmente se realiza mediante una operación *DELETE* en una base de datos.

Existen algunos otros comandos o métodos, como *PATCH* (similar a *PUT*, se usa para cambios parciales), *HEAD* (similar a *GET*, pero para obtener solo los encabezados de la respuesta), etc. Pero se centrará la atención en los cuatro métodos más populares explicados anteriormente.

Por tanto, si se identifica la URI del recurso y el método de la solicitud del cliente, se puede aislar la respuesta deseada y enviarla de vuelta al cliente. Esta respuesta generalmente se proporciona en un formato determinado (generalmente formato JSON o formato XML en algunas aplicaciones antiguas). Ten en cuenta que, para algunos comandos, se debe proporcionar en la solicitud información adicional. Por ejemplo, en los comandos POST será necesario enviar la información que se añadirá al servidor. Se verá más adelante cómo hacer esto.

Se utilizará el *framework* ***Express*** para crear un servidor web Node y aislar cada par de elementos del URI-métodos provenientes de las solicitudes del cliente, para enviar la información correspondiente en formato JSON.

8.3. Primeros pasos con *Express*

8.3.1. ¿Qué es Express?

Express es un *framework* que permite definir un servidor web de una forma realmente sencilla y modular. También permite crear un servidor web utilizando las funcionalidades principales de Node, pero con esta capa adicional proporcionada por *Express*, la información de las solicitudes de los clientes se puede analizar fácilmente y las respuestas a los clientes se pueden tratar por separado y de forma modular. Con *Express* se pueden implementar tanto servidores web estáticos (es decir, servidores que proporcionan documentos web estáticos, hechos con HTML, CSS y Javascript) como proveedores de servicios web. Esto es lo que se hará en esta sesión. Puedes obtener más detalles sobre este *framework* y su documentación oficial en su sitio web.

8.3.2. Instalación de *Express*

Para instalar *Express* en una aplicación Node, se seguirán los mismos pasos que en cualquier módulo de Node anterior: creando el archivo *"package.json"* con *npm init* (si aún no está creado) e instalando el módulo con *npm install*:

```
npm install express
```

Hecho esto, se podrá requerir el módulo en el archivos fuente:

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

8.3.3. Primer ejemplo

Crea un proyecto llamado ***"ExpressTest"*** en el espacio de trabajo. Instala el módulo *Express* siguiendo las instrucciones anteriores y crea un archivo fuente llamado *"server.js"* en la carpeta raíz del proyecto. Luego, escribe las siguientes líneas de código:

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

let app = express();

app.get('/test', (req, res) => {
    res.send('Hello from test URI');
});

app.listen(8080);
```

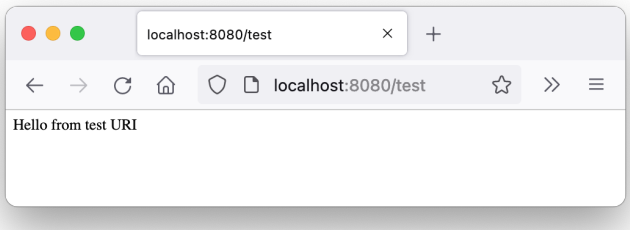
En primer lugar, se ha requerido la biblioteca *Express* y después se inicializa con la función *express()*. A continuación, se define un método *get()* para responder al método GET con el *URI/test*, de modo que si conectamos a esta URI se obtendrá el texto "Hello from test URI" como respuesta. Este método tendrá dos parámetros:

- El URI asociado a este método.
- La devolución de llamada para procesar la solicitud. Esta devolución de llamada es una función que se llama automáticamente cada vez que algún cliente intenta acceder a la URI. La función en sí tiene dos parámetros: *req* (con información de la solicitud) y *res* (un objeto de respuesta para enviar datos al cliente) .

Finalmente, se lanza el servidor escuchando en el puerto 8080. Entonces, si se ejecuta esta aplicación Node (con el comando *node server.js*), podrás abrir un navegador web e ir a la siguiente URL:

```
http://localhost:8080/test
```

Entonces se obtendrá la respuesta esperada:



Si quieres asociar más URIs con un comando GET, solo necesitas definir un método para cada URI. Por ejemplo:

```
app.get('/books', (req, res) => { ... });
app.get('/users', (req, res) => { ... });
```

Eso es todo (por ahora). Hasta este punto has aprendido cómo instalar *Express* y cómo usarlo para implementar un servidor web básico para responder a las solicitudes de servicios web básicos. En la siguiente sección se verá cómo usar *Mongoose* y MongoDB para acceder a datos reales y enviarlos al cliente en formato JSON.

Ejercicio 6

Cree un proyecto llamado ***"Exercise_BasicExpress"*** en tu espacio de trabajo. Instala *Express* y utilízalo para definir un servidor web que responda a estos URIs a través de un comando GET:

URI/date: el servidor enviará al cliente la fecha y hora actual, en cualquier comprensible formato. Es posible que debas usar la biblioteca *"moment"* para completar esta tarea, aunque también puedes confiar en las funciones de fecha estándar de Javascript.

URI/user: el servidor enviará al cliente el usuario que inició sesión en el sistema. Es posible que debas utilizar la biblioteca *"os"* para completar esta tarea.

9. Uso de *Express* con *Mongoose* (I)

Ahora se pasará al siguiente nivel. Ya sabes cómo implementar un servidor *Express* básico, pero se necesita acceder a datos reales para procesar conexiones de clientes reales. Para hacer esto, se utilizarán los servidores MongoDB y la biblioteca *Mongoose*, vistos en la sesión anterior.

Para empezar, se utilizará la base de datos de contactos y se verá cómo listar, añadir, eliminar o actualizar contactos a través de las solicitudes de los clientes.

9.1. Configurar el servidor

En primer lugar, crea una carpeta llamada ***"Exercise_ContactsExpress"*** en tu espacio de trabajo. Instala *Express* y *Mongoose* con los comandos correspondientes *npm init* y *npm install*. En este caso, puedes instalar ambos módulos a la vez con un solo comando:

```
npm install mongoose express
```

A continuación, crea un archivo fuente llamado *"server.js"* y añade el siguiente código:

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

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

let contactSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        minlength: 1,
        trim: true
    },
    telephone: {
        type: String,
        required: true,
        trim: true,
        match: /^\d{9}$/
    },
    age: {
        type: Number,
        min: 18,
        max: 120
    }
});

let Contact = mongoose.model('contact', contactSchema);
let app = express();

app.listen(8080);
```

Como puedes notar, se empieza usando el esquema de contacto simple (sin el esquema *"Address"*), solo para hacerlo simple al principio. En el código anterior, se conecta al servidor Mongo y se lanza el servidor *Express*, sin métodos adicionales. Si intentas ejecutar el servidor (junto con el servidor MongoDB), no obtendrás ninguna respuesta del servidor con ningún URI y método entrante.

Ejercicio 7

Crea el proyecto ***"Exercise_ContactsExpress"*** con las bibliotecas *Express* y *Mongoose* instaladas y copia el código anterior en el archivo fuente *"server.js"*.

9.2. Añadir servicios GET

En primer lugar, se crearán algunas URIs asociadas con un comando GET, para que se pueda obtener información de la base de datos de contactos. Comienza definiendo la lista básica completa, y luego se filtrará un contacto por *id*.

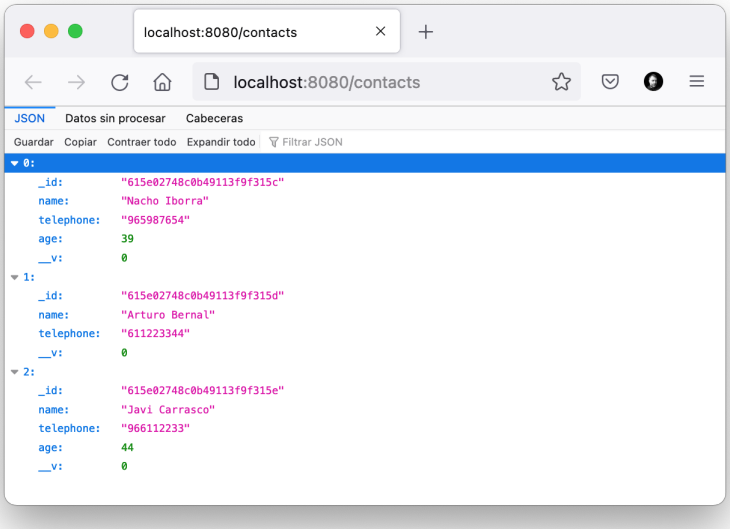
9.2.1. Contacto lista completa

Para obtener la lista completa de contactos, se necesita añadir un nuevo método al servidor *Express* para escuchar una URI dada (se puede llamar */contacts*) con un comando o método GET. Dentro de este método, solo se obtendrá la lista completa de contactos y se enviará al cliente en formato JSON:

```
app.get('/contacts', (req, res) => {
    Contact.find().then(result => {
        res.send(result);
    })
});
```

Si escribes la siguiente URL en un navegador web, verás toda la lista de contactos:

```
http://localhost:8080/contacts
```



La respuesta que obtiene el cliente del servidor dependerá del tipo de contenido de la respuesta en sí. En este caso, como se está enviando un objeto Javascript a través del método ***res.send()***, *Express* lo convierte automáticamente en un objeto JSON, por lo que no hay que preocuparse por él.

9.2.2. Filtrado de datos. URI dinámicos

¿Qué pasa si se quiere obtener la información sobre un contacto específico? Se necesita una forma de enviar al servidor el contacto deseado para obtener sus datos. Hay dos formas de hacer esto:

- Añadir partes dinámicas dentro de la URI. En este caso, se añadiría otro elemento a la URI especificando el *id* del contacto. Por ejemplo:

  ```
  http://localhost:8080/contacts/231982y09ysya9s7ya
  ```

- Añadir parámetros adicionales a la URI (datos de formulario). En este caso, la identificación del contacto no está incrustada en la URI, sino que se añade a la cadena de consulta (más allá del símbolo '?' que determina el final de la URI). El ejemplo anterior se enviará con este formato si se elige esta opción del cliente:

  ```
  http://localhost:8080/contacts?id=231982y09ysya9s7ya
  ```

En ambos casos, el servidor *Express* necesita analizar la URI y extraer la información de identificación. Pero si eliges la primera opción, esto se puede hacer fácilmente sin ayuda externa, mientras que si se usa la cadena de consulta para enviar la información, entonces, es posible que se necesite instalar o usar módulos adicionales que analicen esta cadena de consulta para recuperar la información de la cadena de consulta. Así que centraremos la atención en la primera opción.

En primer lugar, se añadirá un nuevo método al código. En este caso, se esperará una solicitud GET solicitando la URI */contacts/:id*, donde *:id* determinará una parte variable en esta URI, que se asocia automáticamente a una colección llamada *request.params*, con el índice *'id'*.

```
app.get('/contacts/:id', (req, res) => {

});
```

Entonces, se puede simplemente llamar al método *findById* para obtener el resultado asociado a esta identificación y enviarlo de vuelta al cliente. Incluso se puede comprobar si el resultado está definido, y si no, enviar una respuesta adecuada al cliente:

```
app.get('/contacts/:id', (req, res) => {
    Contact.findById(req.params.id).then(result => {
        if (result) {
            let data = {error: false, result: result};
            res.send(data);
        } else {
            let data = {
                error: true,
                errorMessage: "Contact not found"
            };
            res.send(data);
        }
    }).catch(error => {
        let data = {
            error: true,
```

```
            errorMessage: "Error getting contact"
        };
        res.send(data);
    });
});
```

Ten en cuenta que se está creando un objeto Javascript con tres campos:

- ***error***, que determina si la solicitud se ha procesado correctamente (*false*) o no (*true*).
- ***errorMessage***, que está presente si (y solo sí) *error* es *true*. Almacena el mensaje de error que explica lo que salió mal.
- ***result***, que está presente sí (y solo sí) *error* es *false*. Contiene el resultado que tiene encontrado en la colección.

Si la llamada a *findById* funciona correctamente, se entrará en la sección *then* y se comprueba si hay algún resultado válido. Si es así, se envía *error* = *false* con ese resultado. Si no, se envía un *error* = *true* con el mensaje de error. Si algo falla y se lanza una excepción, se entra en la sección de captura y se envía el error correspondiente al cliente. Así es como funcionan las promesas de Javascript.

Ejercicio 8

Actualiza el ejercicio anterior añadiendo estos dos métodos para procesar ambas solicitudes GET. Pruébalos desde un navegador web. Con respecto al segundo método, usa una identificación válida de tu base de datos de contactos para obtener toda la información al respecto.

9.3. Agregar datos con solicitudes POST

En esta subsección, se añadirá un método para manejar las solicitudes POST para insertar contactos en la base de datos.

9.3.1. Obteniendo datos de la solicitud, biblioteca *body-parser*

Dentro de este método, será necesario procesar el cuerpo de la solicitud, del cual se obtendrán los datos de contacto para ser insertados. Esto se puede hacer fácilmente con una biblioteca adicional llamada ***body-parser***. Instálalo en tu aplicación con *npm install*:

```
npm install body-parser
```

Después, añádelo junto con el resto de bibliotecas…

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

y asocialo a con tu servidor *Express* después de crearlo. Al hacer esto, también se indica el tipo de contenido que se analizará desde el cuerpo (contenido JSON, en este caso):

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

Ahora, estás listo para procesar los datos del cuerpo de la solicitud. Pero, te habrás fijado que *bodyParser.json* aparece como *deprecated*. Puedes utilizar la siguiente línea sin necesidad de requerir *body-parser*.

```
app.use(express.json());
```

Obtener datos del cuerpo e insertarlos en Mongo

A continuación, se debe añadir un nuevo método al servidor para procesar las solicitudes POST a la URI */contacts*:

```
app.post('/contacts', (req, res) => {

});
```

Dentro de este método, se procesará el contenido del cuerpo. Se supone que tiene un objeto de contacto en formato JSON, con todos sus campos, para que puede crearse un contacto a partir de él y guardarlo:

```
app.post('/contacts', (req, res) => {
    let newContact = new Contact({
        name: req.body.name,
        telephone: req.body.telephone,
        age: req.body.age
    });
    newContact.save().then(result => {
        let data = {error: false, result: result};
        res.send(data);
    }).catch(error => {
        let data = {error: true, errorMessage:"Error adding contact"};
        res.send(data);
    });
});
```

Tenga en cuenta que se envía la misma estructura de información que antes: los campos *error*, *errorMessage* y *result*. En este caso, si todo está bien, el campo *result* contendrá todo el contacto que se acaba de añadir a la colección.

Ejercicio 9

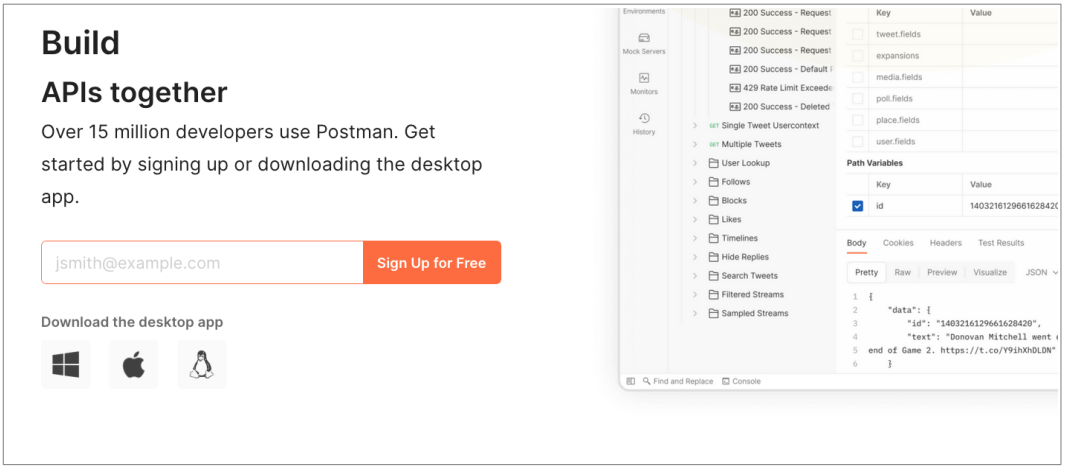
Añade este método al proyecto anterior. ¿Cómo se podría probar si está bien o no? Puedes utilizar un formulario HTML con un método POST, pero los datos no se enviarán en formato JSON. De todos modos, hay una forma más flexible… la respuesta la encontrarás en la siguiente sección.

10. Probando el servidor. *Postman*

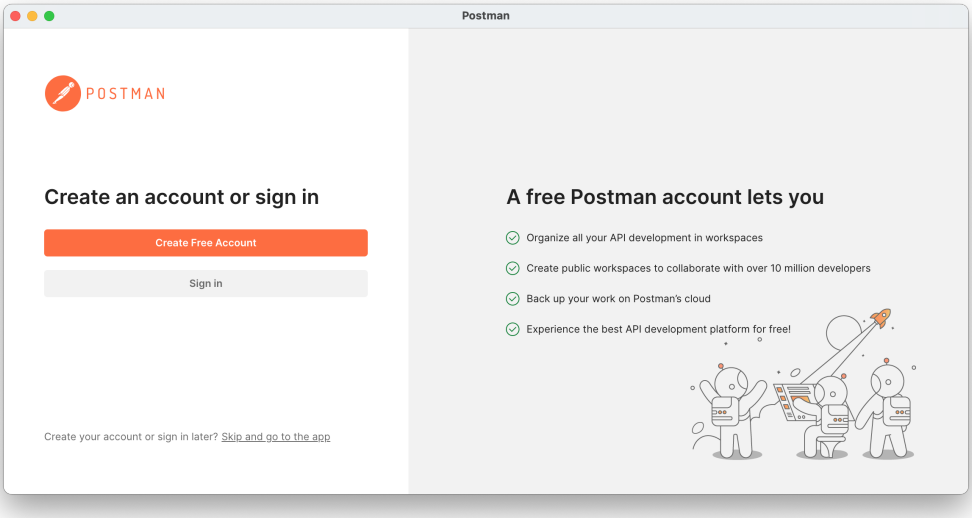
Postman es una aplicación multiplataforma, y gratuita, que permite enviar todo tipo de solicitudes de clientes a un servidor determinado, y obtener la respuesta correspondiente para ver si todo funciona bien antes de utilizar la aplicación cliente en real.

10.1. Descarga, instalación y primeros pasos

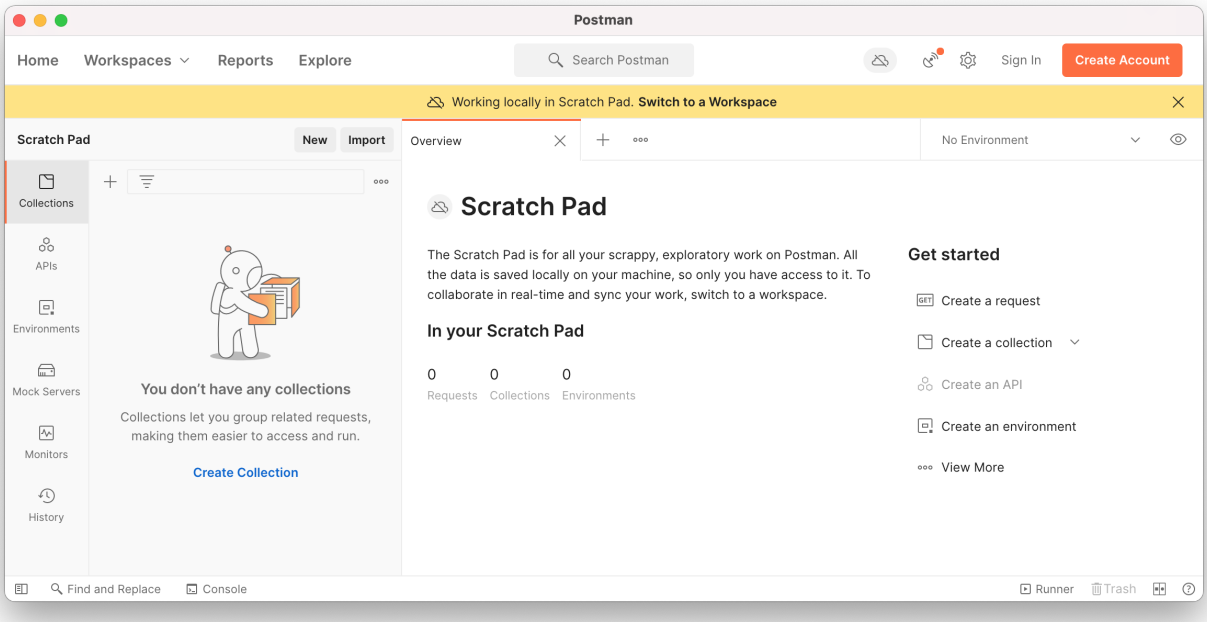
Para descargar e instalar *Postman*, simplemente tienes que ir a su sitio web, y descargar la aplicación, debes elegir el enlace de acuerdo con su sistema operativo:



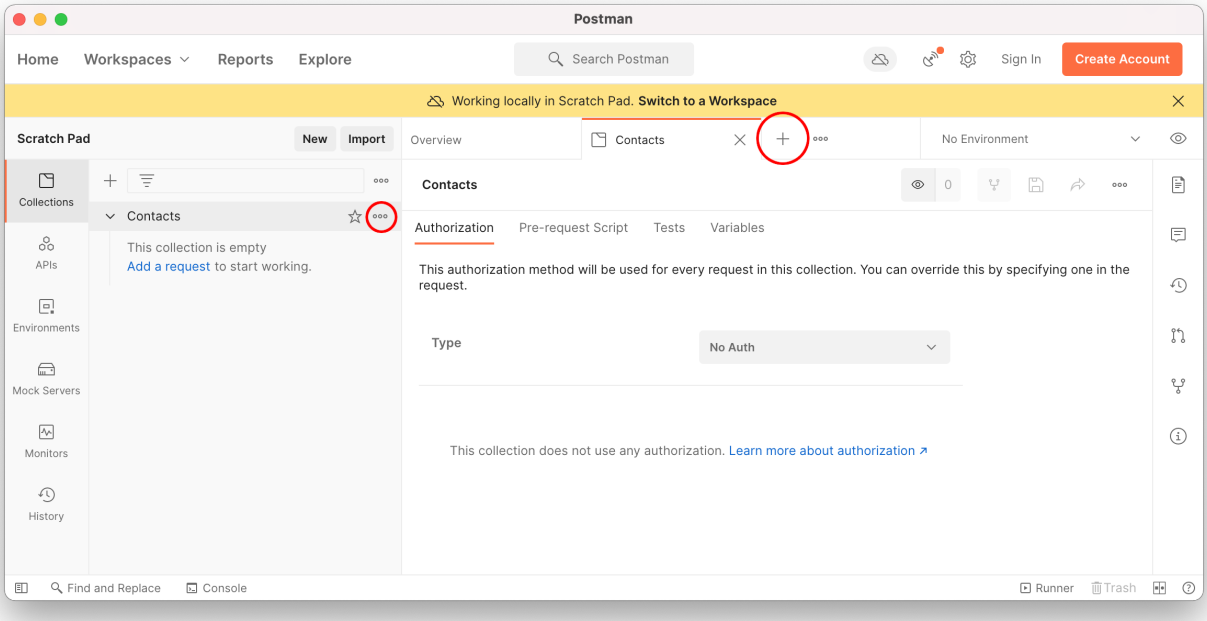
Obtendrás un archivo comprimido, o un ejecutable para Windows. Si lo descomprimimos, podrás ejecutar *Postman* directamente. Al principio, se pedirá que te registres, para poder almacenar todas las pruebas que se hagan con *Postman*, pero este paso no es obligatorio, simplemente puedes omitirlo haciendo clic en el enlace de la parte inferior.



Una vez iniciada la aplicación, verás un diálogo para crear una solicitud simple o una colección (conjunto de solicitudes para una aplicación determinada).



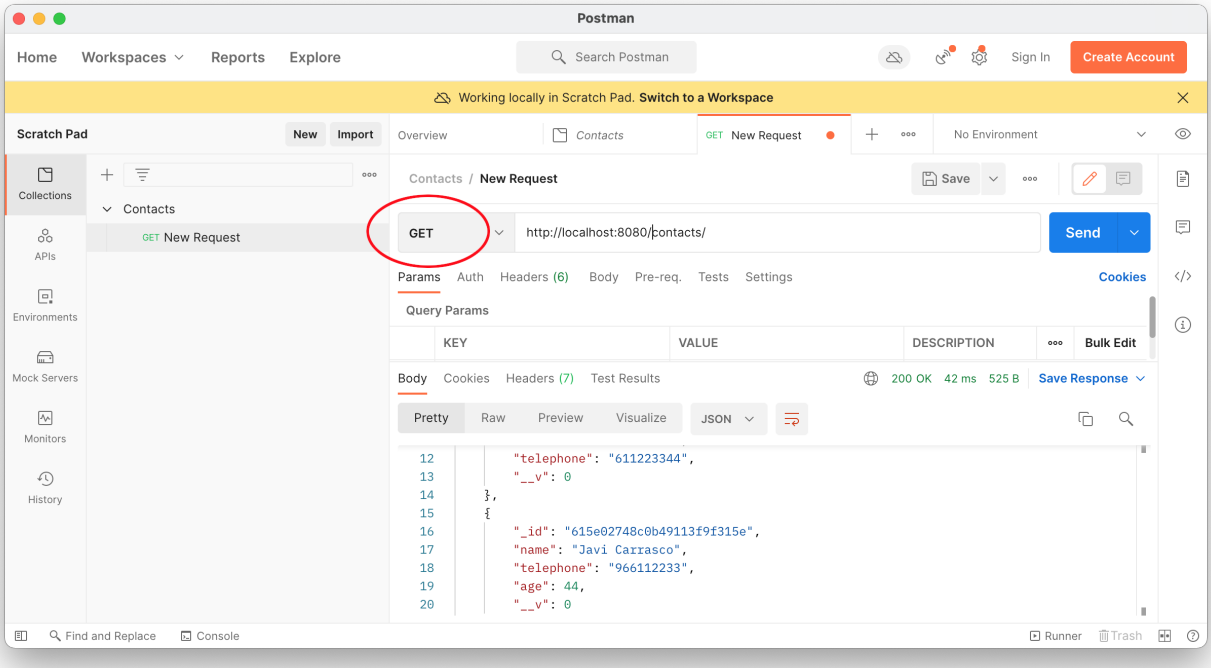
Puedes crear una colección con el nombre deseado (por ejemplo, *"Contacts"*) y guardarla. Se verá en la lista de la izquierda:



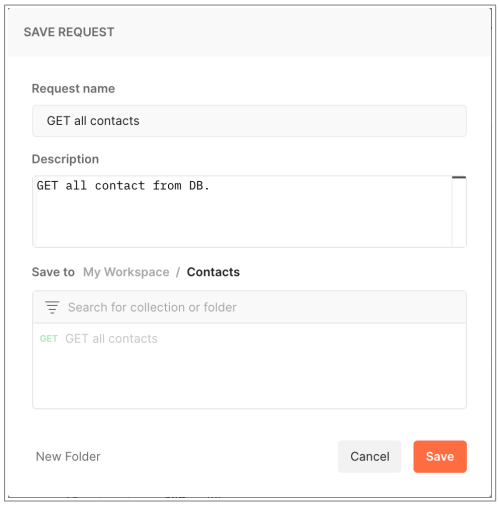
Desde el botón con tres puntos (···) junto a la estrella en la parte izquierda, puedes crear nuevas solicitudes (*add request*) para añadir a esta colección (y también nuevas colecciones), pero hay una manera más fácil de definir las solicitudes a una colección. Simplemente añade una nueva pestaña en el panel central y completa la información de la solicitud.

10.2. Añadir solicitudes GET simples

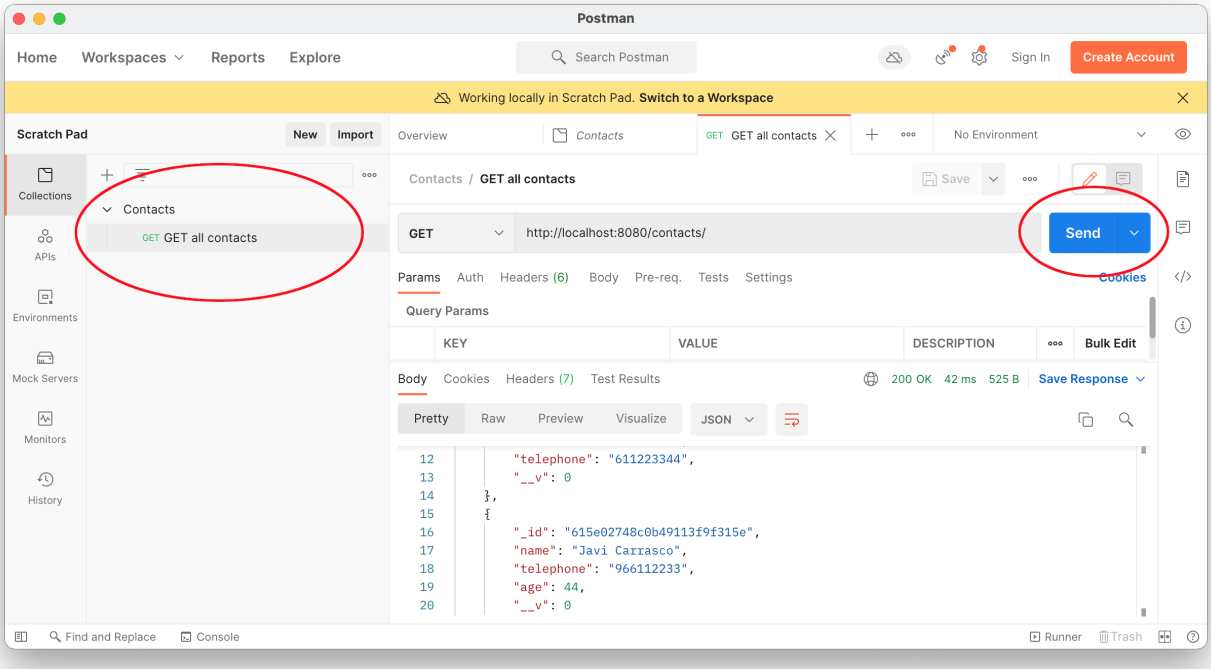
Normalmente, cambia el comando o método (GET, POST, PUT, DELETE) y la URL asociada. Por ejemplo:



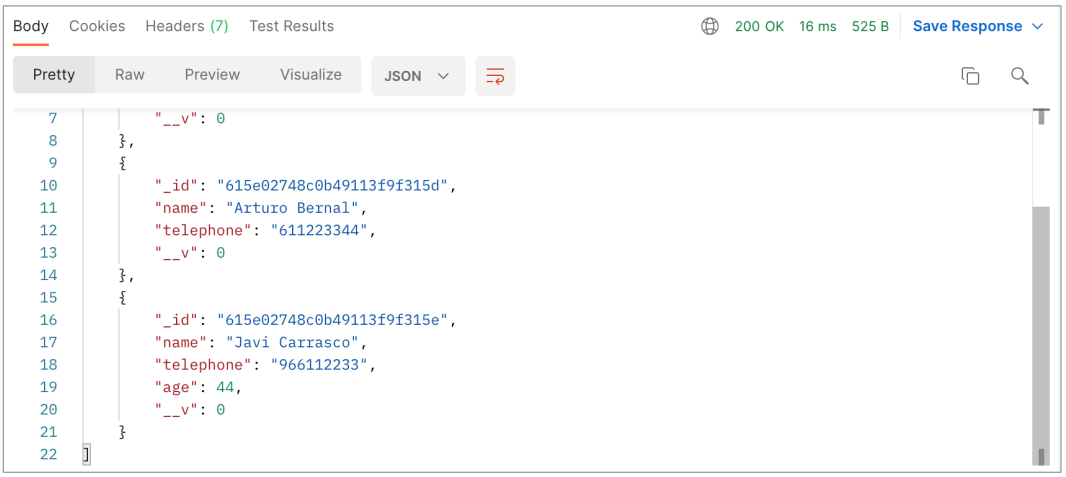
Una vez seleccionado el método, puedes guardar mediante el botón *"Save"* que aparece a la derecha, pero debes proporcionar información para guardar esta solicitud, un nombre, editando el campo que aparece como *"New Request"*. Puedes guardar más información haciendo uso de la opción *"Save As"* en el desplegable del botón. Simplemente establece el nombre de la solicitud (por ejemplo, "GET all contacts", y la colección en la que se almacenará (la colección *"Contacts"*). Luego, puedes hacer clic en el botón Guardar:



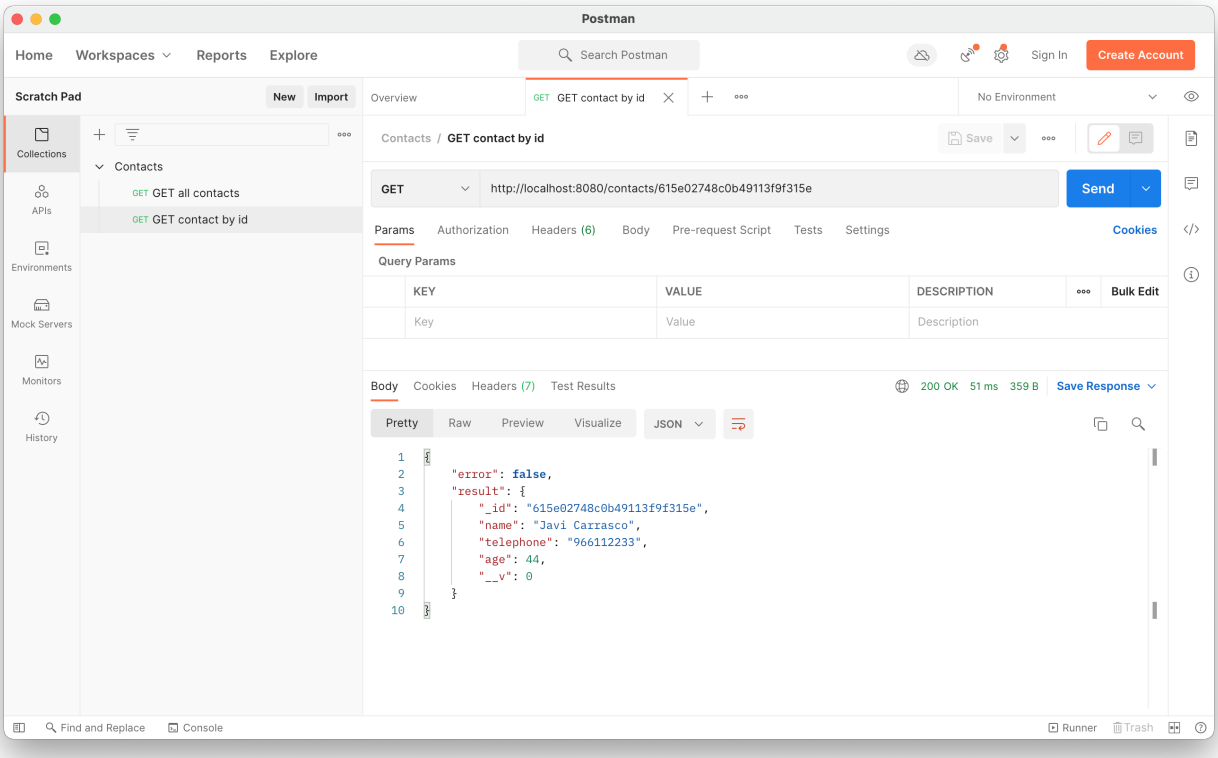
Ahora, puedes seleccionar esta nueva solicitud desde el panel izquierdo, y ejecutarla cuando quieras con el botón *"Send"*.



Cuando se envíe la solicitud, podrás ver la respuesta en el panel inferior:

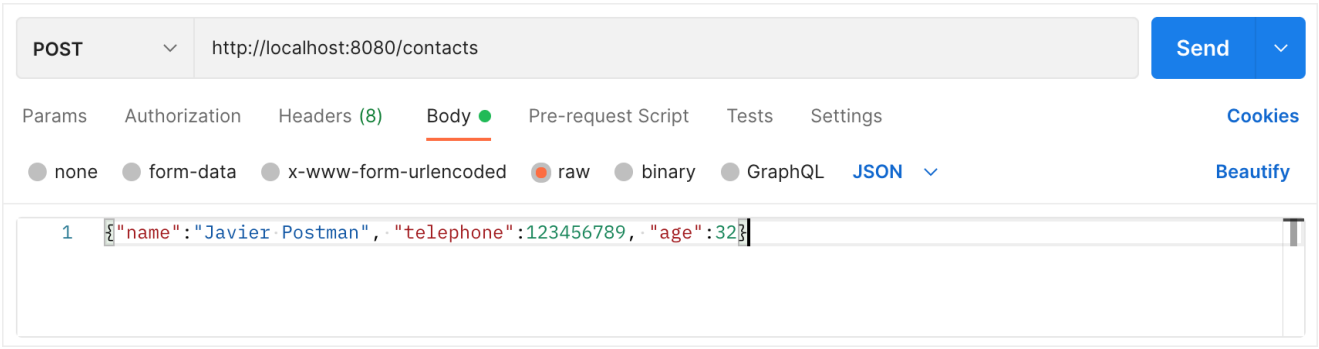


Con estos mismos pasos, también se puede añadir una nueva solicitud GET para obtener un contacto desde su id:



10.3. Añadir solicitudes POST

Las solicitudes POST son diferentes de las solicitudes GET, ya que también envían algunos datos en el cuerpo de la solicitud. ¿Cómo puede hacerse esto con *Postman*? Crea una nueva solicitud en la colección *"Contacts"*, de tipo POST, conectándose a la URI */contacts*. Después, haz clic en la pestaña *Body* debajo de la URL y configura el tipo como *raw*. También puedes cambiar la propiedad *Text* a la derecha a JSON (*application/json*) para que puedas ver la sintaxis resaltada al añadir los datos JSON.



En este caso, se está enviando un nuevo contacto con nombre "Javier Postman", teléfono "123456789" y edad 32. Se puede guardar y enviar esta solicitud, obteniendo el resultado correspondiente del servidor Node (siempre y cuando lo tengamos funcionando, por supuesto :-)

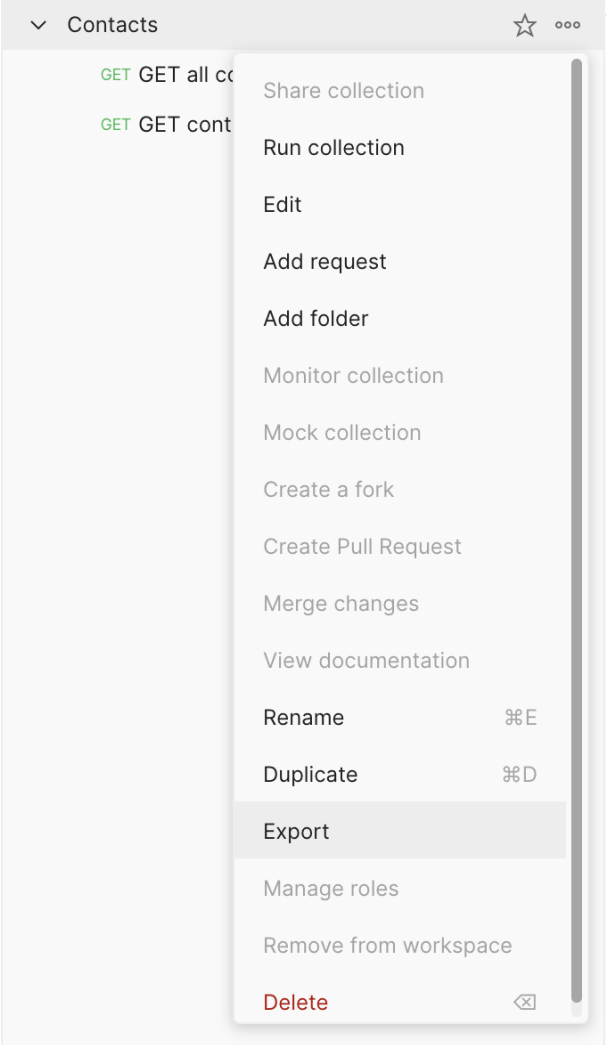
Ejercicio 10

Crea una nueva colección de *Postman* llamada *"Contacts"* (si aún no la ha creado) y añade las solicitudes que se han visto antes:

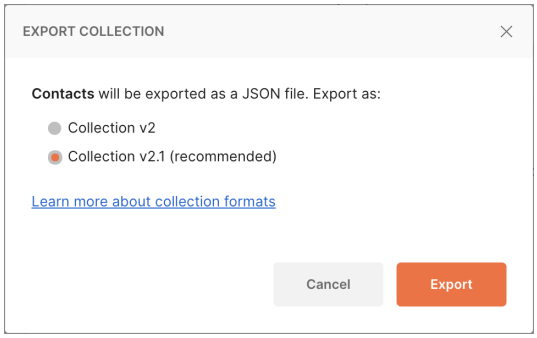
- GET all contacts
- GET a contact by its id
- POST a new contact

10.4. Exportar/Importar colecciones

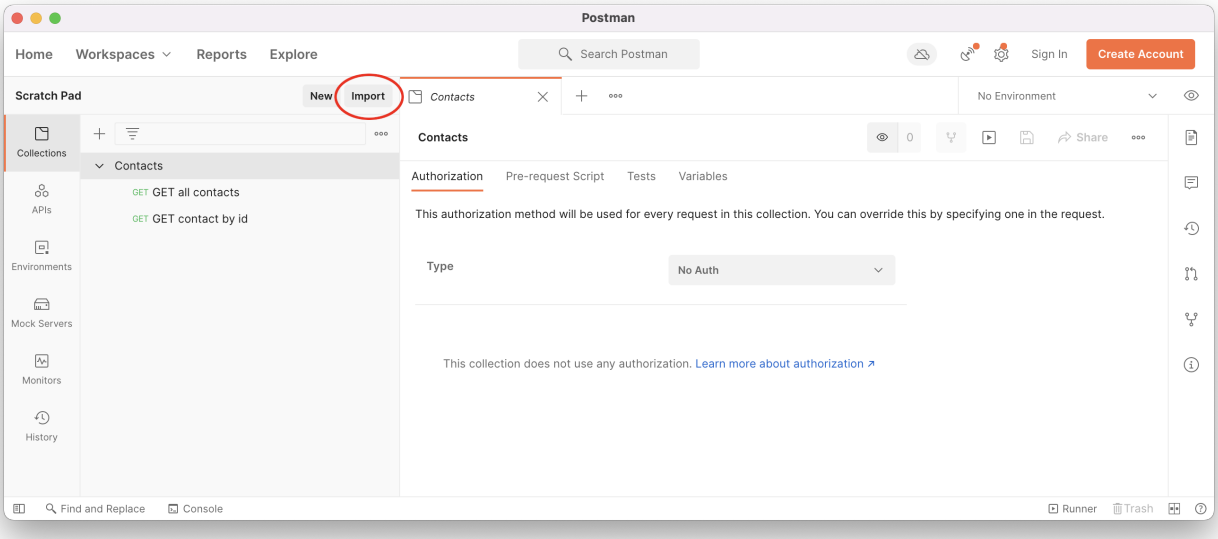
También se pueden exportar e importar colecciones de solicitudes en *Postman*. Para exportar una colección, haz clic en el botón con tres puntos (···) junto al nombre de la colección y selecciona *Export*.



Se pedirá que configures la versión de *Postman* para exportar (la recomendada es una buena opción):



Después, se creará un nuevo archivo *Postman* en la carpeta seleccionada. Si deseas importar una colección determinada, haz clic en el botón Importar en la esquina superior izquierda de ventana principal de *Postman*:



Después, elige el archivo *Postman* y la colección se cargará en el panel izquierdo.

11. Uso de *Express* con *Mongoose* (II)

Ahora que ya sabes cómo usar *Postman* para crear todo tipo de solicitudes, se terminará con los dos tipos de solicitudes que se han dejado de lado: PUT y DELETE.

11.1. Actualización de datos con solicitudes PUT

Si quieres actualizar los datos de un documento dado, necesitas enviar un comando PUT. Como se hizo con el método POST, se necesitan enviar algunos datos a través del cuerpo de la solicitud, por lo que será necesario usar la biblioteca *body-parser* para procesar esta información. Además, se especificará el *id* del contacto a actualizar en la URI, por lo que se trabajará con este patrón:

```
/contacts/:id
```

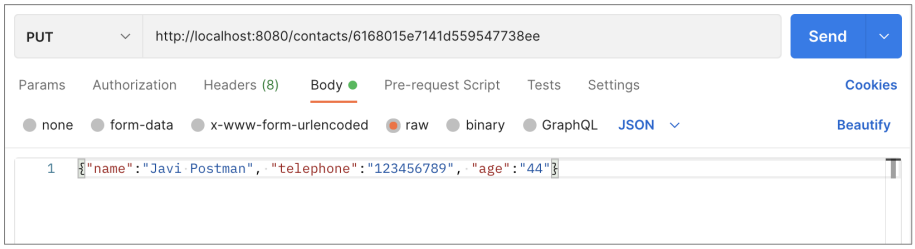
Echa un vistazo al código del nuevo método del servicio:

```
app.put('/contacts/:id', (req, res) => {
    Contact.findByIdAndUpdate(req.params.id, {
        $set: {
            name: req.body.name,
            telephone: req.body.telephone,
            age: req.body.age
        }
    }, {new: true}).then(result => {
        let data = {error: false, result: result};
        res.send(data);
    }).catch(error => {
        let data = {error: true, errorMessage:"Error updating contact"};
        res.send(data);
    });
});
```

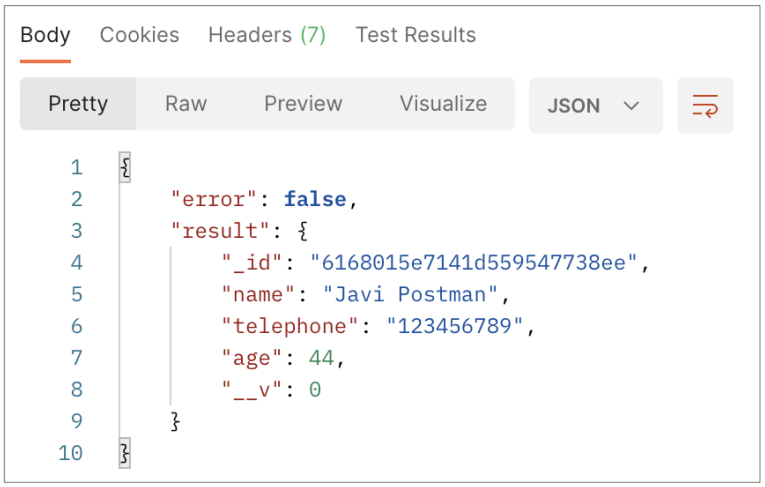
Como puedes ver, se obtiene la identificación de los parámetros de la solicitud, y luego se busca el contacto por su identificación para actualizar a través del método *findByIdAndUpdate()*. Se establecen todos los atributos nuevos (obteniendo los nuevos del cuerpo) y se procesa el resultado.

11.1.1. Añadir una solicitud PUT en *Postman*

Para probar este nuevo método (después de reiniciar el servidor Node), necesitarás añadir una nueva solicitud a la colección *Postman*. En este caso, es una solicitud PUT para la URI */contacts/:id* (asegúrese de usar una identificación existente). En el Cuerpo de la solicitud, configura los nuevos datos para este contacto:



Si se envía esta solicitud al servidor Node, podrás ver la correspondiente respuesta de éxito:



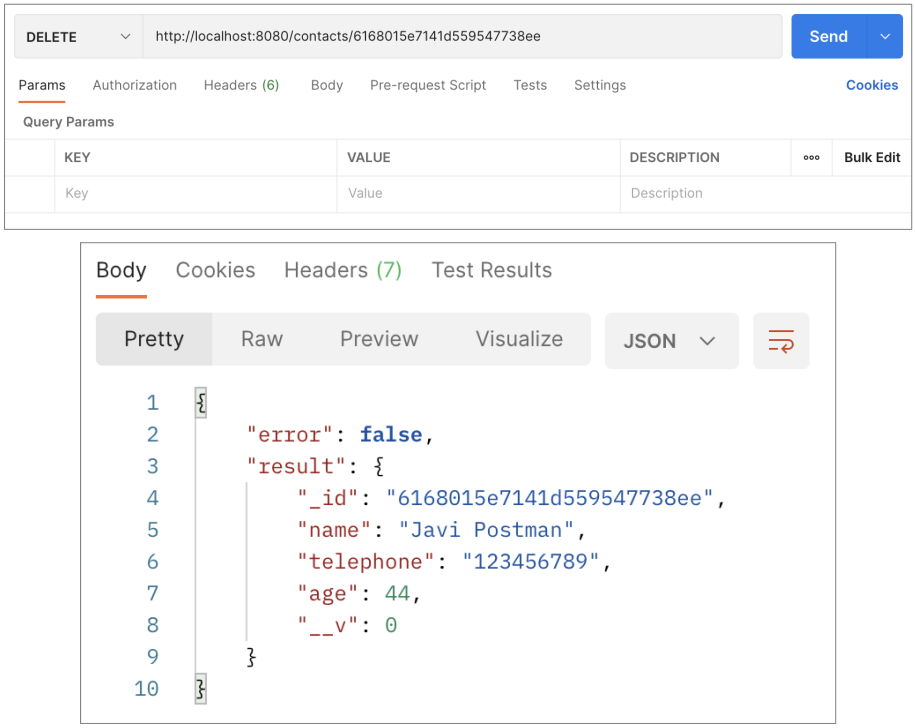
11.2. Eliminar datos con solicitudes DELETE

Finalmente, añade el último método importante al servidor. Para eliminar contactos de la colección, se llamará a la URI */contacts/:id* con un comando DELETE. Dentro del método del servidor, se llamará al método *findByIdAndRemove()* y se procesará la promesa:

```
app.delete('/contacts/:id', (req, res) => {
    Contact.findByIdAndRemove(req.params.id).then(result => {
        let data = {error: false, result: result};
        res.send(data);
    }).catch(error => {
        let data = {error: true, errorMessage:"Error removing contact"};
        res.send(data);
    });
});
```

11.2.1. Añadir una nueva solicitud DELETE a *Postman*

Si deseas añadir una solicitud DELETE a la colección, es muy similar a una solicitud GET, especificando el método y la URI con el *id* del elemento a eliminar, luego podrás enviar la solicitud y obtener el resultado:



Ejercicio 11

Añade estos dos nuevos métodos a tu servidor Node y sus correspondientes solicitudes de *Postman* para probarlos.

Ejercicio 12

Cree un proyecto llamado ***"Exercise_BooksExpress"*** e intenta implementar un servidor Node con el esquema de libros explicado para el ejercicio 5 de la sesión anterior. Añade las siguientes URIs con sus métodos correspondientes para añadir/eliminar/actualizar/mostrar libros de la colección:

- ***GET /books*** para enumerar todos los libros.
- ***GET /books/:id*** para obtener la información de un libro determinado.
- ***POST /books*** para insertar un libro nuevo.
- ***PUT /books/:id*** para actualizar los datos de un libro.
- ***DELETE /books/:id*** para eliminar un libro de la colección.

Si también trabajas con el esquema de *Author*, es posible que tengas que añadir un servicio adicional:

- ***POST /authors*** para insertar nuevos autores, de modo que puedan vincularse a los libros.