5. Desarrollo de servicios REST con Express

5.1. Introducción. Servicios REST

5.1.1. 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 una línea 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. Dependiendo de cómo se haya implementado el servidor, también podríamos reescribir esta URL de este otro modo, con el mismo significado:

http://miservidor.com/libros/123

5.1.2. 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, no obstante, en los cuatro métodos principales anteriores

Por lo tanto, identificando el recurso a solicitar (URI) 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 último 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.

5.1.3. 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}]
```

JavaScript ofrece un par de métodos útiles para convertir datos a formato JSON y viceversa. Estos métodos son `JSON.stringify` (para convertir un objeto o array JavaScript a JSON) y `JSON.parse` (para el proceso inverso, es decir, convertir una cadena JSON en un objeto JavaScript). Aquí vemos un ejemplo de cada uno:

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

// Convertir array a JSON
let personasJSON = JSON.stringify(personas);
console.log(personasJSON);

// Convertir JSON a array
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.

5.1.3.1. JSON y servicios REST

Como comentábamos antes, JSON es hoy en día el formato más utilizado para dar respuesta a peticiones de servicios REST. Su otro "competidor", el formato XML, está cada vez más en desuso para estas tareas.

A la hora de emitir una respuesta a un servicio utilizando formato JSON, es habitual que ésta tenga un formato determinado. En general, en las respuestas que emitamos a partir de ahora para servicios REST en el curso, utilizaremos una estructura general basada en:

- Un dato booleano (podemos llamarlo `ok`, por ejemplo), que indique si la petición se ha podido atender satisfactoriamente o no.
- Un mensaje de error (podemos llamarlo `error`, por ejemplo), que estará presente únicamente si el anterior dato booleano es falso, lo que indicaría que la petición no se ha podido resolver.
- Los datos de respuesta, que estarán presentes sólo si el dato booleano es verdadero, lo que indica que la petición se ha podido atender satisfactoriamente. Notar que estos datos de respuesta pueden ser un texto, un objeto simple JavaScript, o un array de objetos.

Adicionalmente, como veremos en los ejemplos a continuación, también es recomendable añadir a la respuesta un código de estado HTTP, que indique si se ha podido servir satisfactoriamente o ha habido algún error.

5.2. ¿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.

5.2.1. 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` en la carpeta del proyecto donde queramos añadirlo (y, previamente, el comando `npm init` en el caso de que aún no hayamos creado el archivo *package.json*):

```
npm install express
```

5.2.1.1. Ejemplo de servidor básico con Express

Vamos a crear un proyecto llamado "*PruebaExpress*" en la carpeta de "*ProyectosNode/Pruebas*", y a instalar Express en él. Después, creamos un archivo llamado `index.js` con este código:

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

El código, como podemos ver, 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 poner en marcha la aplicación Node, abrir un navegador y acceder a la URL:

http://localhost:8080

Si pruebas a ejecutar la aplicación Node desde Visual Studio Code, 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, por lo que dará un mensaje de error al intentar acceder a cualquier URL. Esto lo solucionaremos en los en los siguientes apartados.

5.2.2. 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);
```

5.2.2.1. 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 o URI 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.

5.2.3. 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.

5.2.3.1. 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.
- ...

5.2.3.2. 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 (información detrás del interrogante `?` en una URL)
- `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
- ...

5.2.3.3. 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 o parámetro `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)
- `render(vista, [opciones])` : para mostrar una determinada vista como respuesta, pudiendo especificar opciones adicionales y un callback de respuesta.
- ...

5.3. 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*"), continuación del proyecto *PruebaContactosMongo_v2* de sesiones anteriores (copia y pega el código de ese proyecto en esta nueva carpeta). 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 una carpeta `models` con los modelos de datos (contactos, restaurantes y mascotas, según habíamos creado en versiones anteriores) y un archivo `index.js`, que anteriormente lanzaba una serie de operaciones simples, y donde ahora vamos a definir nuestro servidor Express con las rutas para responder a las diferentes operaciones sobre los contactos.

5.3.1. 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, incorporar los modelos de datos, conectar con la base de datos y poner en marcha el servidor Express:

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

const Contacto = require(__dirname + "/models/contacto");
const Restaurante = require(__dirname + "/models/restaurante");
const Mascota = require(__dirname + "/models/mascota");

mongoose.connect('mongodb://localhost:27017/contactos',
    {useNewUrlParser: true, useUnifiedTopology: true});

let app = express();
app.listen(8080);
```

5.3.2. Servicios de listado (GET)

5.3.2.1. 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, usando el modelo Mongoose que ya hemos creado. Devolveremos el resultado directamente en la respuesta, lo que lo convertirá automáticamente a formato JSON. Añadimos el servicio en el archivo principal `index.js`. Normalmente se añaden antes de poner en marcha el servidor (después de haber creado la variable `app`).

```
app.get('/contactos', (req, res) => {
    Contacto.find().then(resultado => {
        res.status(200)
           .send( {ok: true, resultado: resultado});
    }).catch (error => {
        res.status(500)
           .send( {ok: false,
                   error: "Error obteniendo contactos");
    });
});
```

Observad que enviamos un código de estado (200 si todo ha ido bien, 500 o fallo del servidor si no hemos podido recuperar los contactos), y el objeto JSON con los campos que explicábamos antes: el dato booleano indicando si se ha podido servir o no la respuesta, y el mensaje de error o el resultado correspondiente, según sea el caso.

5.3.2.2. 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.status(200)
               .send({ok: true, resultado: resultado});
        else
            res.status(400)
               .send({ok: false,
                      error: "No se han encontrado contactos"});
    }).catch (error => {
        res.status(400)
           .send({ok: false,
                  error: "Error buscando el contacto indicado"});
    });
});
```

En este caso, distinguimos si el objeto `resultado` obtenido con `findById` devuelve algo o no, para emitir una u otra respuesta. En caso de que no se pueda encontrar el resultado, asumimos que es a causa de que la petición del cliente no es correcta, y emitimos un código de estado 400 (por ejemplo).

5.3.2.3. 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
    });
});
```

En cualquier caso, en estos apuntes optaremos por la versión anterior, incluyendo el parámetro en la propia URI.

5.3.2.4. Prueba de los servicios desde el navegador

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* 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".

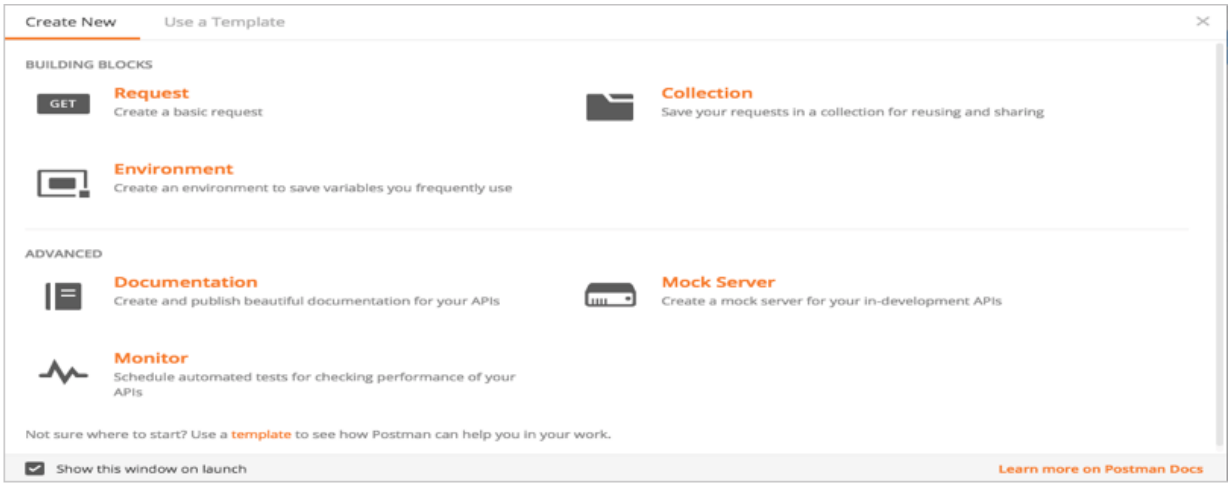
5.4. Uso de Postman para probar servicios REST

Ya hemos visto que probar unos servicios de listado (GET) es sencillo a través de un navegador. Sin embargo, pronto aprenderemos a hacer otros servicios (inserciones, modificaciones y borrados) que no son tan sencillos de probar con esta herramienta. Así que conviene ir entrando en contacto con otra más potente, que nos permita probar todos los servicios que vamos a desarrollar. Esa herramienta es Postman.

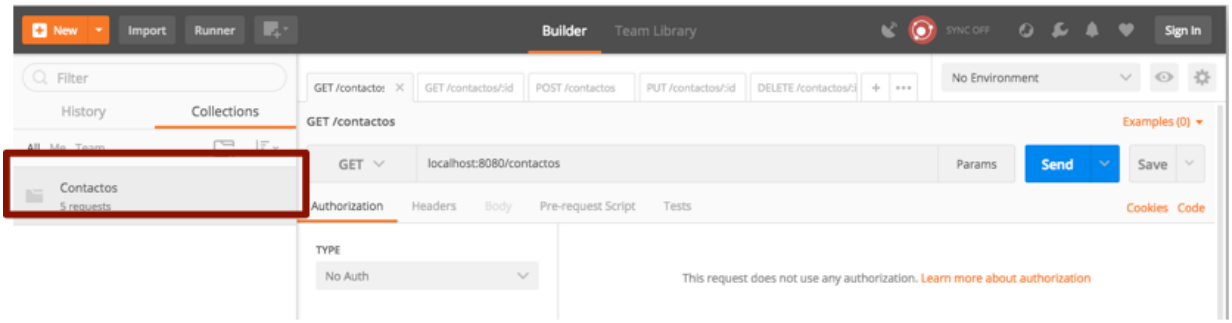
Postman es una aplicación gratuita y multiplataforma que permite enviar todo tipo de peticiones de clientes a un servidor determinado, y examinar la respuesta que éste produce. De esta forma, podemos comprobar que los servicios ofrecen la información adecuada antes de ser usados por una aplicación cliente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

En el documento de instalación se explica cómo instalar y poner en marcha Postman en nuestro equipo, y si estás utilizando la máquina virtual con todo el software instalado, ya lo tendrás listo para funcionar.

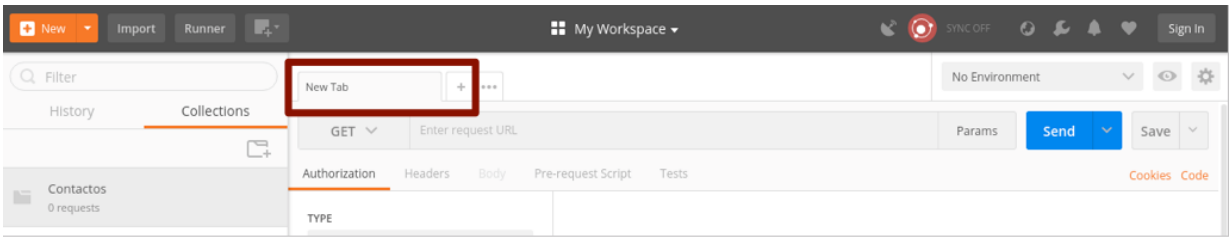
Tras iniciar la aplicación, veremos un diálogo para crear peticiones simples o colecciones de peticiones (conjuntos de pruebas para una aplicación). Lo que haremos habitualmente será esto último.



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:

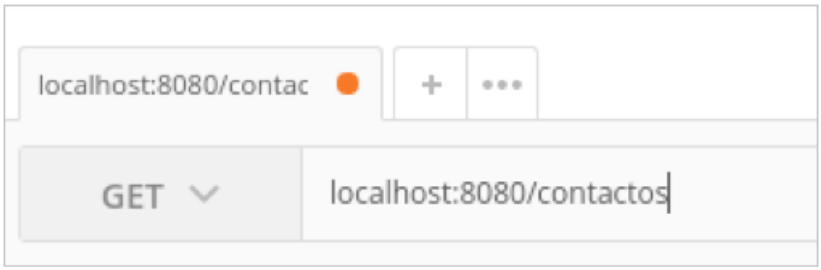


Desde el botón "New" en la esquina superior izquierda podemos crear nuevas peticiones (también nuevas colecciones) y asociarlas a una colección. Existe una forma alternativa (quizá más cómoda) de crear esas peticiones, a través del panel de pestañas, añadiendo nuevas:

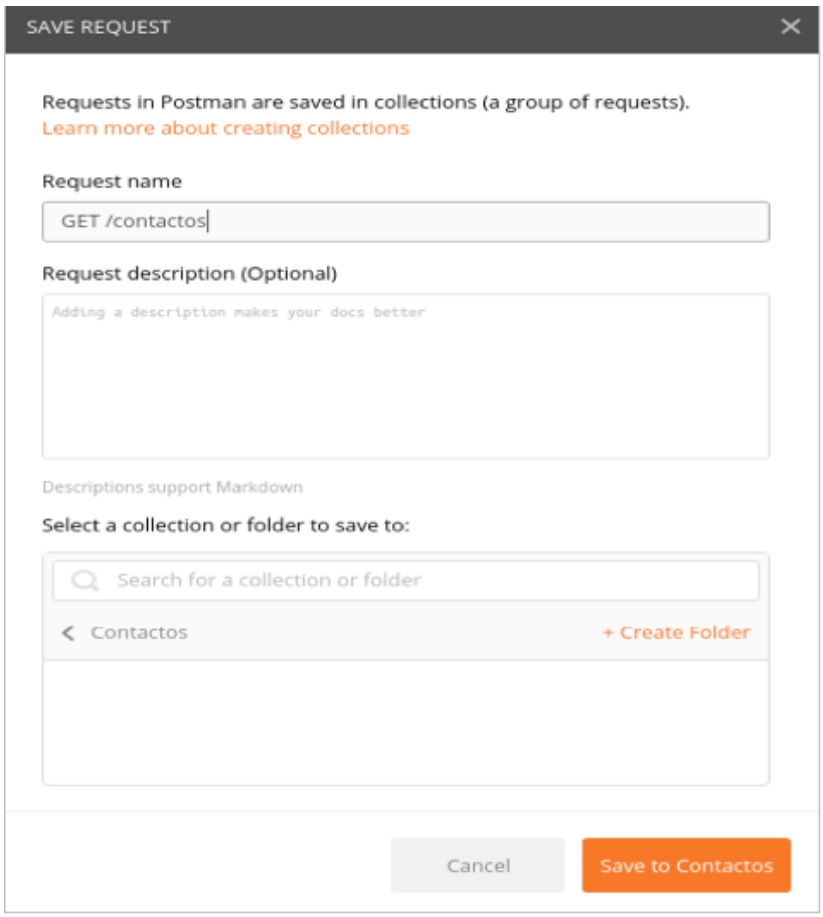


5.4.1. Añadir peticiones GET

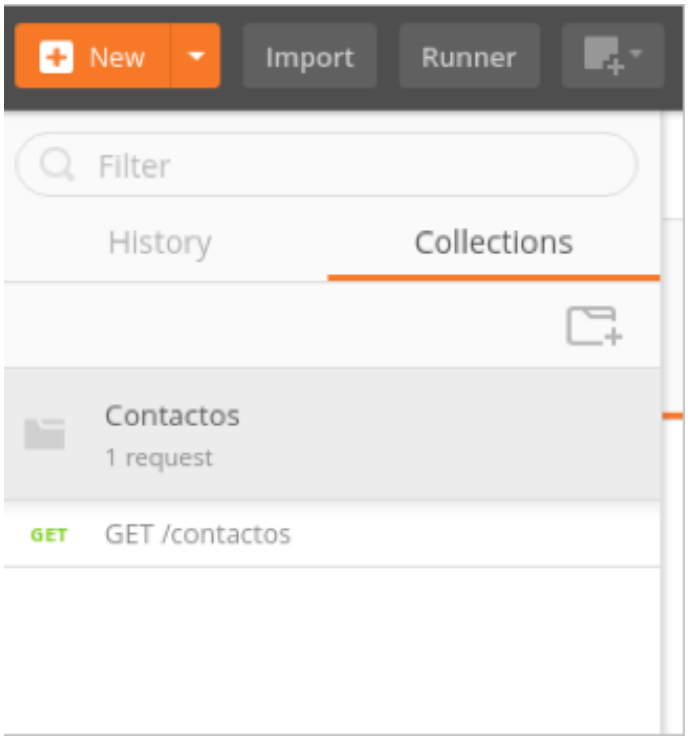
Para añadir una petición, habitualmente elegiremos el tipo de comando bajo las pestañas (GET, POST, PUT, DELETE) y la URL asociada a dicho comando. Por ejemplo:



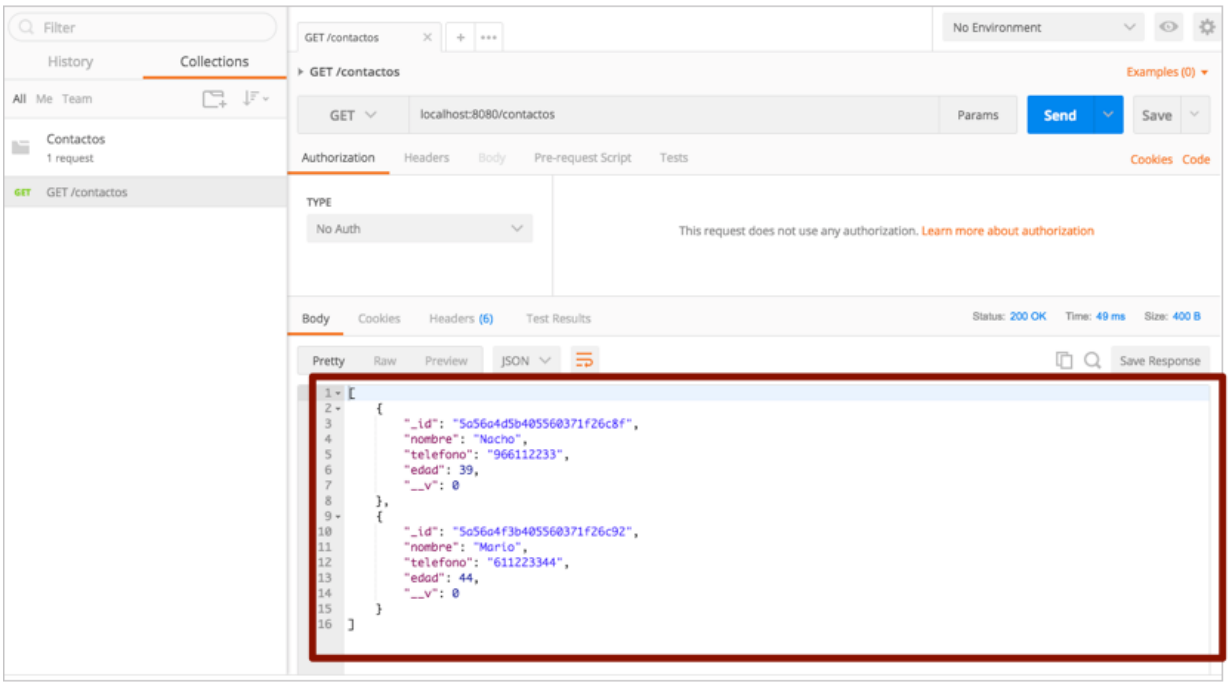
Entonces, podemos hacer clic en el botón "Save" en la parte derecha, y guardar la petición para poderla reutilizar. Al guardarla, nos pedirá que le asignemos un nombre (por ejemplo, "GET /contactos" en este caso), y la colección en la que se almacenará (nuestra colección de "Contactos").



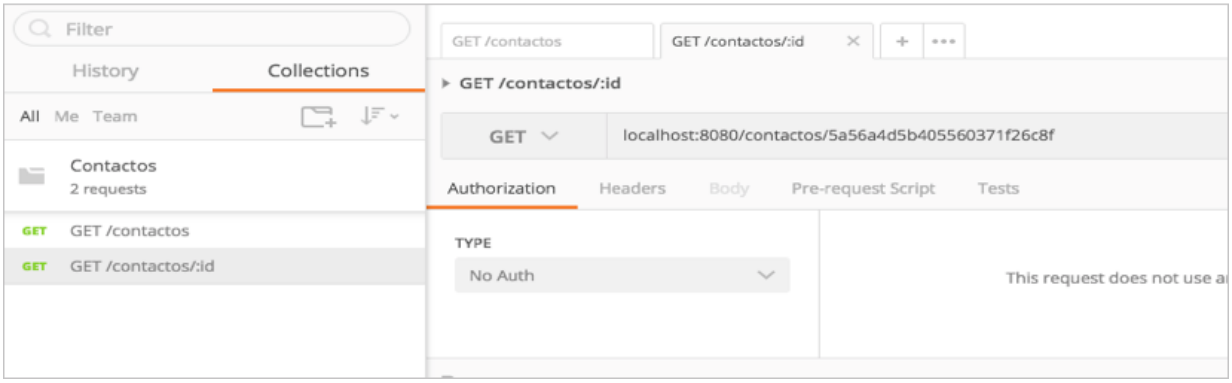
Después, podremos ver la prueba asociada a la colección, en el panel izquierdo:



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:

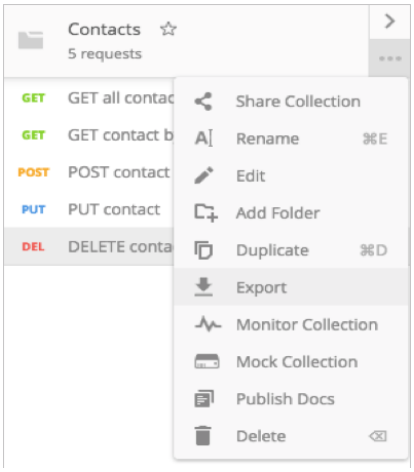


Bastaría con reemplazar el *id* de la URL por el que queramos consultar realmente. Si probamos esta petición, obtendremos la respuesta correspondiente:

5.4.2. Exportar/Importar colecciones

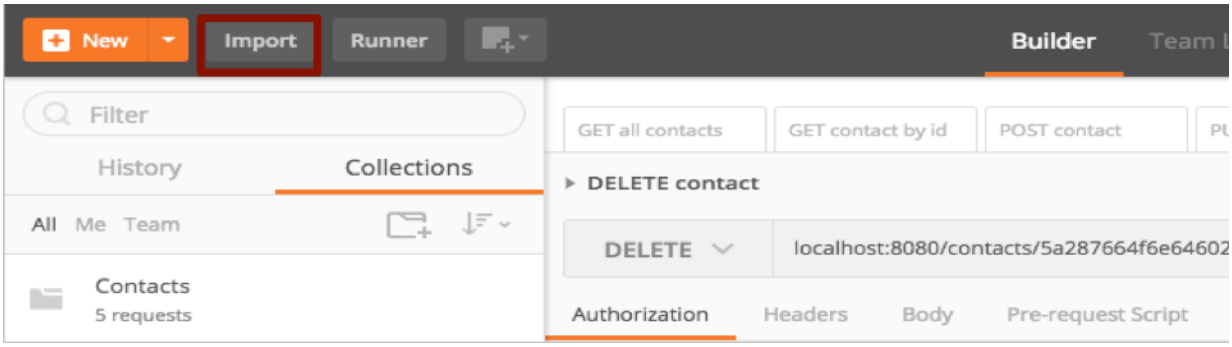
Podemos exportar e importar nuestras colecciones en Postman, de forma que podemos llevarlas de un equipo a otro. Para **exportar** una colección, hacemos clic en el botón de puntos suspensivos (...) que hay junto a ella

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, y elegir el archivo (previamente exportado):



5.5. Operaciones de actualización (POST, PUT, DELETE)

Tras haber visto cómo añadir servicios GET a un servidor Express, quedan pendientes las otras tres operaciones básicas (POST, PUT y DELETE), así que veremos ahora cómo desarrollarlas en Express, y cómo probarlas con la herramienta Postman.

5.5.1. Las inserciones (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. Para poder recoger esta información desde el cliente en nuestro servidor Node/Express, debemos utilizar un *middleware* que procese la petición para dejarnos accesibles y preparados estos datos.

Express incorpora, desde su versión 4.16, un middleware propio para este cometido. Basta con añadirlo a la aplicación, justo después de inicializar la `app` Express. Como lo que vamos a hacer es trabajar con objetos JSON, añadiremos el procesador JSON de este modo:

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

> **NOTA**: antes de la versión 4.16, para este cometido era necesario utilizar una librería de terceros llamada *body-parser*. Aquí tenéis la documentación sobre dicha librería.

Ahora vamos a nuestro servicio POST. Añadimos para ello un método `post` del objeto `app`, con los mismos parámetros que tenía el método `get` (recordemos: la ruta a la que responder, y el *callback* que se ejecutará como respuesta). El método en cuestión podría quedar así:

```
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.status(200)
           .send({ok: true, resultado: resultado});
    }).catch(error => {
        res.status(400)
           .send({ok: false,
                  error: "Error añadiendo contacto"});
    });
});
```

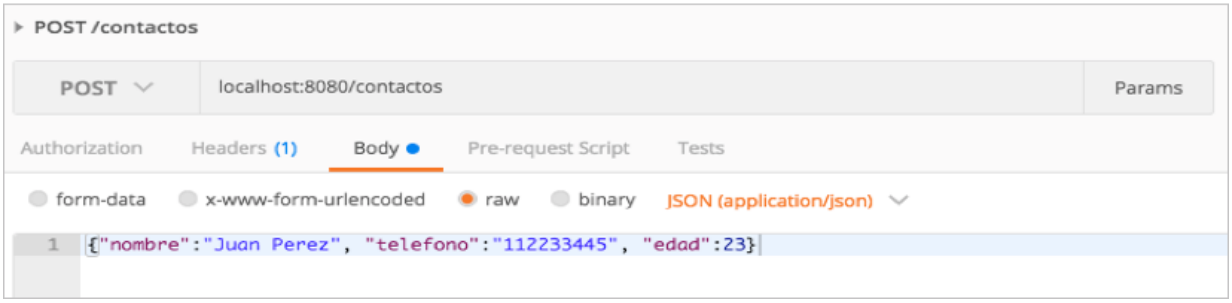
Al principio, construimos el contacto a partir de los datos JSON que llegan en el cuerpo, accediendo a cada campo por separado con `req.body.nombre_campo`. Esto es posible hacerlo gracias a que el *middleware* anterior ha pre-procesado la petición, y nos ha dejado los datos disponibles dentro del objeto `req.body`. De lo contrario, tendríamos que leer los bytes de la petición manualmente, almacenarlos en un array, y convertir desde JSON.

El resto del código es el que ya conoces de ejemplos previos con Mongoose (llamada a `save` y procesamiento del resultado), combinado con el envío del código de estado y la respuesta REST.

5.5.1.1. Prueba de operaciones POST con Postman

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 *middleware* 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:



5.5.2. Las modificaciones (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*, normalmente), y utilizaremos el mismo *middleware* anterior 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. El código puede ser similar a éste:

```
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.status(200)
           .send({ok: true, resultado: resultado});
    }).catch(error => {
        res.status(400)
           .send({ok: false,
                  error:"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 del *middleware* de Express para procesar la

petición y obtener los datos que llegan en formato JSON. Tras la llamada al método, devolvemos el estado y la respuesta JSON correspondiente.

5.5.2.1. Prueba de operaciones PUT con Postman

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:

5.5.3. Los borrados (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. Obtendremos dicho *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.status(200)
           .send({ok: true, resultado: resultado});
    }).catch(error => {
        res.status(400)
           .send({ok: false,
                  error:"Error eliminando contacto"});
    });
});
```

5.5.3.1. Prueba de operaciones DELETE con Postman

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:

5.5.4. 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 `if..else` si el resultado es correcto o no, y mostrar una u otra cosa:

```
if (resultado)
    res.status(200)
       .send({ok: true, resultado: resultado});
else
    res.status(400)
       .send({ok: false,
              error: "No se ha encontrado el contacto"});
```

En este punto puedes realizar el Ejercicio 2 de los propuestos al final de la sesión.

5.5.5. Más sobre el *middleware* de procesado de la petición

Existen otras posibilidades de uso del *middleware* que procesa la petición. 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. Por ejemplo:

```
nombre=Nacho&telefono=911223344&edad=39
```

Para procesar contenidos de este otro tipo, basta con cargar el *middleware* de este otro modo:

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

También es posible añadir ambos modos juntos:

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

En este caso, el servidor Express aceptaría datos de la petición tanto en formato JSON como en formato *query-string*. En cualquier caso, deberemos asegurarnos desde el cliente (incluso si usamos Postman) 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.

5.6. Estructurando una API REST en Express

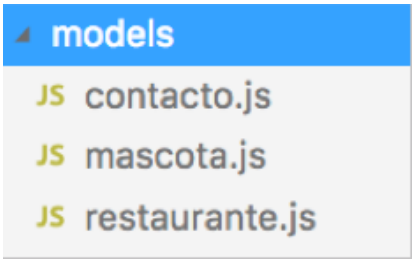
Los ejemplos hechos hasta ahora de aplicaciones Express como proveedor de servicios REST son bastante monolíticos: en un solo archivo fuente hemos ubicado la aplicación Express en sí y las rutas a las que responderá.

A pesar de que el propio framework Express se define en su web oficial como *unopinionated*, es decir, sin opinión acerca de cómo debe ser una arquitectura de aplicación Express, sí conviene seguir ciertas normas mínimas de modularidad en nuestro código. Consultando ejemplos en Internet podemos encontrar distintas formas de estructurar aplicaciones Express, y podríamos considerar correctas muchas de ellas, desde el punto de vista de modularidad del código. Aquí vamos a proponer una estructura que seguir en nuestras aplicaciones, basándonos en otros ejemplos vistos en Internet, pero que no tiene por qué ser la mejor ni la más universal.

Para empezar, crearemos una copia de nuestro proyecto *PruebaContactosExpress* en otro llamado *PruebaContactosExpress_v2*, donde iremos incorporando los cambios que veremos a continuación.

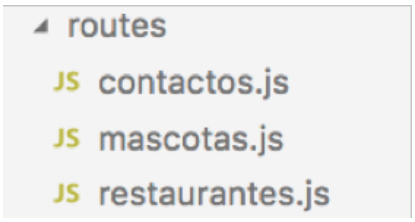
5.6.1. Los modelos de datos

Es habitual encontrarnos con una carpeta `models` en las aplicaciones Express donde se definen los modelos de las diferentes colecciones de datos. En nuestro ejemplo de contactos, dentro de esa carpeta "models" ya hemos definido los archivos para nuestros tres modelos de datos: `contacto.js`, `restaurante.js` y `mascota.js`, y los hemos incorporado con `require` desde el programa principal:



5.6.2. Las rutas y enrutadores

Imaginemos que la gestión de contactos en sí (alta / baja / modificación / consulta de contactos) se realizará mediante servicios englobados en una URI que empieza por `/contactos`. Para el caso de restaurantes y mascotas, utilizaremos las URIs `/restaurantes` y `/mascotas`, respectivamente. Vamos a definir tres enrutadores diferentes, uno para cada cosa. Lo normal en estos casos es crear una subcarpeta `routes` en nuestro proyecto, y definir dentro un archivo fuente para cada grupo de rutas. En nuestro caso, definiríamos un archivo `contactos.js` para las rutas relativas a la gestión de contactos, otro `restaurantes.js` para los restaurantes, y otro `mascotas.js` para las mascotas.



> **NOTA**: es también habitual que la carpeta `routes` se llame `controllers` en algunos ejemplos que podemos encontrar por Internet, ya que lo que estamos definiendo en estos archivos son básicamente controladores, que se encargan de comunicarse con el modelo de datos y ofrecer al cliente una respuesta determinada.

Vamos a definir el código de estos tres enrutadores que hemos creado. En cada uno de ellos, utilizaremos el modelo correspondiente de la carpeta "*models*" para poder manipular la colección asociada.

Comencemos por la colección más sencilla de gestionar: la de **mascotas**. Definiremos únicamente servicios para listar (GET), insertar (POST) y borrar (DELETE). El código del enrutador `routes/mascotas.js` quedaría así (se omite el código interno de cada servicio, que sí puede consultarse en los ejemplos de código de la sesión):

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

let Mascota = require(__dirname + '/../models/mascota.js');

let router = express.Router();

// Servicio de listado
router.get('/', (req, res) => {
    ...
});

// Servicio de inserción
router.post('/', (req, res) => {
    ...
});

// Servicio de borrado
router.delete('/:id', (req, res) => {
    ...
});

module.exports = router;
```

Notar que utilizamos un objeto `Router` de Express para gestionar los servicios, a diferencia de lo que veníamos haciendo en sesiones anteriores, donde nos basábamos en la propia aplicación (objeto `app`) para gestionarlos. De esta forma, definimos un router para cada grupo de servicios, que se encargará de su procesamiento. Lo mismo ocurrirá para los dos enrutadores siguientes (restaurantes y contactos).

Notar también que las rutas no hacen referencia a la URI `/mascotas`, sino que apuntan a una raíz `/`. El motivo de esto lo veremos en breve.

De forma análoga, podríamos definir los servicios GET, POST y DELETE para los **restaurantes** en el enrutador `routes/restaurantes.js`:

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

let Restaurante = require(__dirname + '/../models/restaurante.js');

let router = express.Router();

// Servicio de listado
router.get('/', (req, res) => {
    ...
});

// Servicio de inserción
router.post('/', (req, res) => {
    ...
});

// Servicio de borrado
router.delete('/:id', (req, res) => {
    ...
});

module.exports = router;
```

Quedan, finalmente, los servicios para **contactos**. Adaptaremos los que ya hicimos en pasos anteriores, copiándolos en el enrutador `routes/contactos.js`. El código quedaría así:

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

let Contacto = require(__dirname + '/../models/contacto.js');

let router = express.Router();

// Servicio de listado general
router.get('/', (req, res) => {
    ...
});

// Servicio de listado por id
router.get('/:id', (req, res) => {
    ...
});

// Servicio para insertar contactos
router.post('/', (req, res) => {
    ...
});

// Servicio para modificar contactos
router.put('/:id', (req, res) => {
    ...
});

// Servicio para borrar contactos
router.delete('/:id', (req, res) => {
    ...
});

module.exports = router;
```

5.6.3. La aplicación principal

El servidor principal ve muy aligerado su código. Básicamente se encargará de cargar las librerías y enrutadores, conectar con la base de datos y poner en marcha el servidor:

```
// Librerías externas
const express = require('express');
const mongoose = require('mongoose');

// Enrutadores
const mascotas = require(__dirname + '/routes/mascotas');
const restaurantes = require(__dirname + '/routes/restaurantes');
const contactos = require(__dirname + '/routes/contactos');

// Conexión con la BD
mongoose.connect('mongodb://localhost:27017/contactos',
    {useNewUrlParser: true, useUnifiedTopology: true});

let app = express();

// Carga de middleware y enrutadores
app.use(express.json());
app.use('/mascotas', mascotas);
app.use('/restaurantes', restaurantes);
app.use('/contactos', contactos);

// Puesta en marcha del servidor
app.listen(8080);
```

Los enrutadores se cargan como *middleware*, empleando `app.use`. En esa instrucción, se especifica la ruta con la que se mapea cada enrutador, y por este motivo, dentro de cada enrutador las rutas ya hacen referencia a esa ruta base que se les asigna desde el servidor principal; por ello todas comienzan por `/`.

En este punto, puedes realizar el Ejercicio 3 de los propuestos al final de la sesión. También se deja propuesto como opcional el Ejercicio 4.

5.7. Ejercicios propuestos

Para los ejercicios de esta sesión, crea una subcarpeta llamada "**Sesion5**" en tu carpeta "**ProyectosNode/Ejercicios**", para dentro ir creando un proyecto para cada ejercicio.

Ejercicio 1

Crea una carpeta llamada "**Ejercicio_5_1**" en la carpeta de ejercicios "*ProyectosNode/ Ejercicios/Sesion5*". 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 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.

Ejercicio 2

Crea una carpeta llamada "**Ejercicio_5_2**" en la carpeta de ejercicios "*ProyectosNode/ Ejercicios/Sesion5*". Instala Express y Mongoose en ella, y crea un archivo `index.js` que incorpore el modelo de libros básico (sin autores ni comentarios) que habrás hecho en los ejercicios de la sesión 4 para el *Ejercicio_4* (deja ese modelo en el archivo `models/libro.js` del proyecto).

Después crea una instancia de servidor Express, y da respuesta a estos servicios:

- `GET /libros`: devolverá un listado en formato JSON del array de libros completo de la colección.
- `GET /libros/:id`: devolverá un objeto JSON con los datos del libro encontrado a partir de su *id*.
- `POST /libros`: recogerá los datos del libro que le llegarán en el cuerpo de la petición e insertará el libro en cuestión en la base de datos, devolviendo un objeto JSON con el libro insertado.
- `PUT /libros/:id`: recogerá los datos del libro que le llegarán en el cuerpo de la petición, y el *id* del libro a modificar de los parámetros de la URL, y realizará los cambios correspondientes, devolviendo un objeto JSON con el libro modificado.
- `DELETE /libros/:id`: eliminará el libro cuyo *id* se reciba como parámetro en la URL, devolviendo en formato JSON el libro borrado.

En todos los casos, se emitirá un código de estado acorde al resultado de la respuesta, y un objeto JSON con los datos comentados en estos apuntes (un booleano `ok` indicando si se ha atendido bien o no la petición, un campo `error` con el mensaje de error, si lo hay, y el resultado a enviar si la petición ha sido exitosa).

Comprueba el funcionamiento de estos servicios desde Postman, creando una colección llamada **LibrosV1**. Exporta la colección y adjúntala a este proyecto cuando todo funcione correctamente.

Ejercicio 3

Crea una copia del ejercicio anterior en otra carpeta llamada "**Ejercicio_5_3**", y estructura aquí la aplicación tal y como se ha explicado en el apartado 5.6 de esta sesión, separando el modelo de datos, los enrutadores o controladores y la aplicación principal.

Cuando ya tengas la aplicación lista, crea una colección en Postman llamada **LibrosV2**, y añade dentro las pruebas de las peticiones GET, POST, PUT y DELETE para los servicios desarrollados. Comprueba su correcto funcionamiento, y después exporta la colección a un archivo, que adjuntarás a este proyecto.

Ejercicio 4

Opcional

Sobre el ejercicio anterior, añade en las carpetas correspondientes los siguientes archivos:

- El modelo de autores, y su correspondiente relación con el modelo de libros. El propio modelo de autores puedes recuperarlo de los ejercicios de la sesión 4.
- El enrutador para los autores. En este enrutador sólo vamos a definir los servicios de listado general (GET), inserción (POST) y borrado (DELETE).
- Modifica el programa principal para que los libros respondan a URIs con el prefijo `/libros` y los autores estén asociados al prefijo `/autores`.
- Añade las tres pruebas de GET, POST y DELETE para autores a la colección de Libros de Postman del ejercicio anterior.