CRUD - Endpoints

Ahora que tenemos controlados los errores, sabemos crear nuevos endpoints en Express y tenemos control sobre nuestra base de datos, vamos a continuar aprendiendo el proceso CRUD con los endpoints restantes, para terminar nuestra primera API funcional.

Propiedad body para añadir nuevos documentos a nuestra DB

Al igual que los enpoints pueden tener **query y route params,** los endpoints de tipo **POST y PUT** pueden tener una propiedad llamada **body** que utilizaremos para enviar datos a estos endpoints.

Vamos a aprovechar el último proyecto sobre La Casa de Papel, en el que se ha añadido un nuevo atributo al model Character que es role, y a crear un enpoint para añadir nuevos caracteres.

Hasta ahora hemos utilizado las rutas directamente en el `index.js` de nuestro servidor, pero de este momento en adelante vamos a crear una carpeta llamada `routes` en la que exportaremos cada tipo de ruta en un archivo distinto para mejorar nuestro código, en nuestro caso, crearemos el archivo `character.routes.js` en el que añadiremos todas las rutas relacionadas con mascotas.

```
// Archivo character.routes.js dentro de la carpeta routes
const express = require('express');

const Character = require('../models/Character');

const router = express.Router();

router.get('/', async (req, res, next) => {
  try {
    const characters = await Character.find();
    return res.status(200).json(characters)
```

```
  } catch (error) {
    return next(error)
  }
});

module.exports = router;
```

Y ahora, ¿por qué el endpoint utiliza **'/'** en vez **'/characters'** como antes? Vamos a ver el motivo y razonar su solución.

Añadiremos en el archivo `index.js` un nuevo middleware de rutas para el enpoint **'/characters':**

```
//Realizamos los requires
const express = require('express');
const {connect} = require('./utils/db')
const Character = require('./models/Character');
const characterRoutes = require('./routes/character.routes')

connect(); // Llamamos a la función connect que conecta con MongoDB
const PORT = 3000;
const server = express();
const router = express.Router();

server.use('/characters', characterRoutes);

server.use('*', (req, res, next) => {
  const error = new Error('Route not found');
  error.status = 404;
  next(error);
  });

server.use((error, req, res, next) => {
  return res.status(error.status || 500).json(error.message || 'Unexpected error');
});

server.listen(PORT, () => {
  console.log(`Server running in http://localhost:${PORT}`);
});
```

Por tanto, nuestro código se comporta de la siguiente forma:

- El cliente hace una request a `http://localhost:3000/characters`
- El código de index.js llega al punto `server.use('/characters', characterRoutes)` como hay coincidencia realiza una llamada characterRoutes.
- Al entrar en `character.routes.js` ya se ha considerado la parte `/characters` de la ruta, por lo que el archivo empieza en su nueva base `/`**.**
- Se busca el enpoint GET, POST, PUT o DELETE que coincida con la nueva base de ruta y se lanza la función Response o Next para continuar o terminar el proceso en ese controlador, en nuestro caso reconoce una petición get y ejecuta la llamada a la base de datos para devolver todos los personajes.

Ahora que sabemos como funciona la división de rutas por archivos, vamos a crear nuestro endpoint POST para crear personajes:

Utilizando POST para crear un documento de nuestra DB

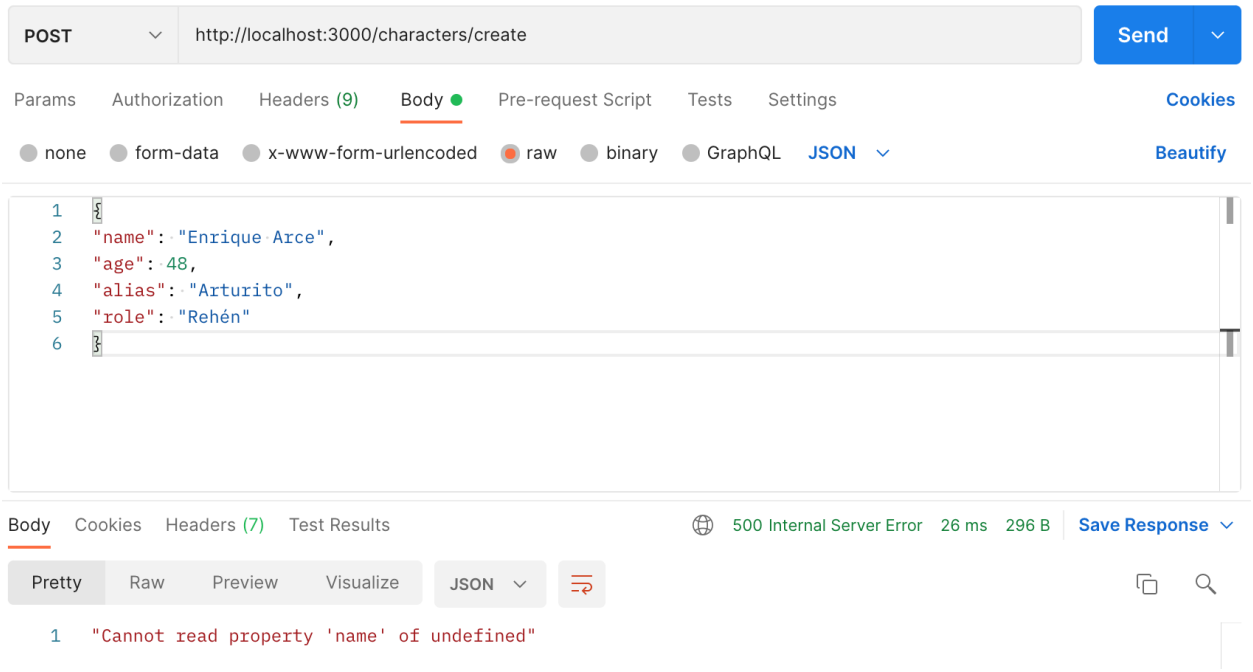
Para crear nuestra primera ruta POST nos situamos en character.routes.js y creamos nuestro endpoint.

```
router.post('/create', async (req, res, next) => {
    try {
      // Crearemos una instancia de character con los datos enviados
      const newCharacter = new Character({
        name: req.body.name,
        age: req.body.age,
        alias: req.body.alias,
        role: req.body.role
      });

      // Guardamos el personaje en la DB
      const createdCharacter = await newCharacter.save();
      return res.status(201).json(createdCharacter);
    } catch (error) {
          // Lanzamos la función next con el error para que lo gestione Express
      next(error);
    }
  });

module.export = router;
```

¿Cómo accedemos a este endpoint? ¿Cómo enviamos información a req.body?

Al endpoint se accederá a través de http://localhost:3000/characters/create pero, no podremos hacerlo a través del navegador todavía, ya que necesitaríamos un formulario o una función **fetch** que envíe datos en formato **JSON**, así que vamos a utilizar **Postman** para simular peticiones.

Vamos a añadir una nueva request POST, con el endpoint que acabamos de crear, y enviaremos los datos en formato `raw` y `JSON` de la siguiente manera:



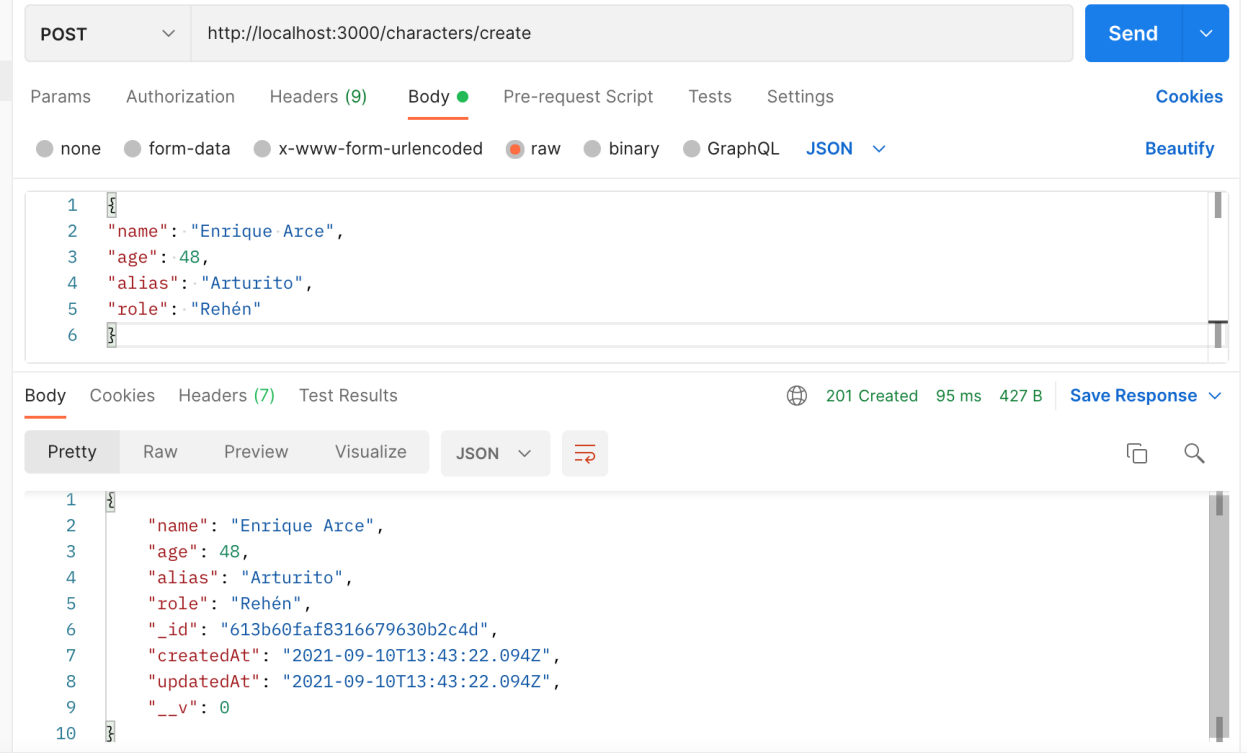
Si tenemos todo configurado correctamente, podremos enviar la petición a través de Postman y deberíamos poder crear el nuevo personaje en la base de datos, pero **¡tendremos un error!**

Esto se debe a que **req.body** no contendrá los datos que hemos enviado al servidor porque no estamos utilizando ningún middleware para "parsear" la información. Para ello, añadiremos estas líneas en el archivo `index.js` antes de añadir las rutas.

Estas son funciones propias de express que transforman la información enviada como JSON al servidor de forma que podremos obtenerla en **req.body.**

```
server.use(express.json());
server.use(express.urlencoded({ extended: false }));
```

Repetiremos el proceso para confirmar que ahora podemos crear nuevos personajes:



👋 ¡Bravo! ya podemos crear nuevos documentos en la base de datos utilizando el método POST.

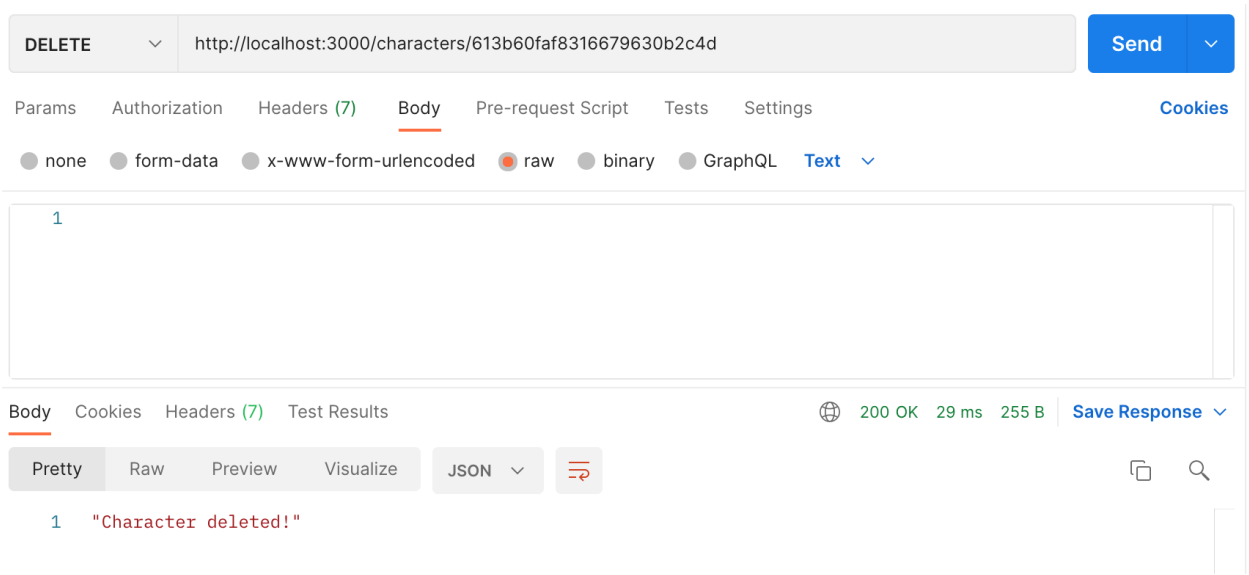
Utilizando DELETE para eliminar un documento de la DB

Aunque podamos encontrarlo en algún proyecto, no es correcto enviar body a nuestro endpoit de tipo DELETE, por lo tanto utilizaremos un parámetro de ruta ***/:id*** para identificar el elemento que queremos eliminar de nuestra DB.

Empezaremos creando un endpoint del tipo delete en nuestro character.routes.js.

```
router.delete('/:id', async (req, res, next) => {
    try {
        const {id} = req.params;
        // No será necesaria asignar el resultado a una variable ya que vamos a eliminarlo
        await Character.findByIdAndDelete(id);
        return res.status(200).json('Character deleted!');
    } catch (error) {
        return next(error);
    }
});
```

Con esto, podremos lanzar la petición al endpoint usando el método DELETE y podremos ver como hemos eliminado el documento de nuestra base de datos.



Utilizando PUT para editar un documento de nuestra DB

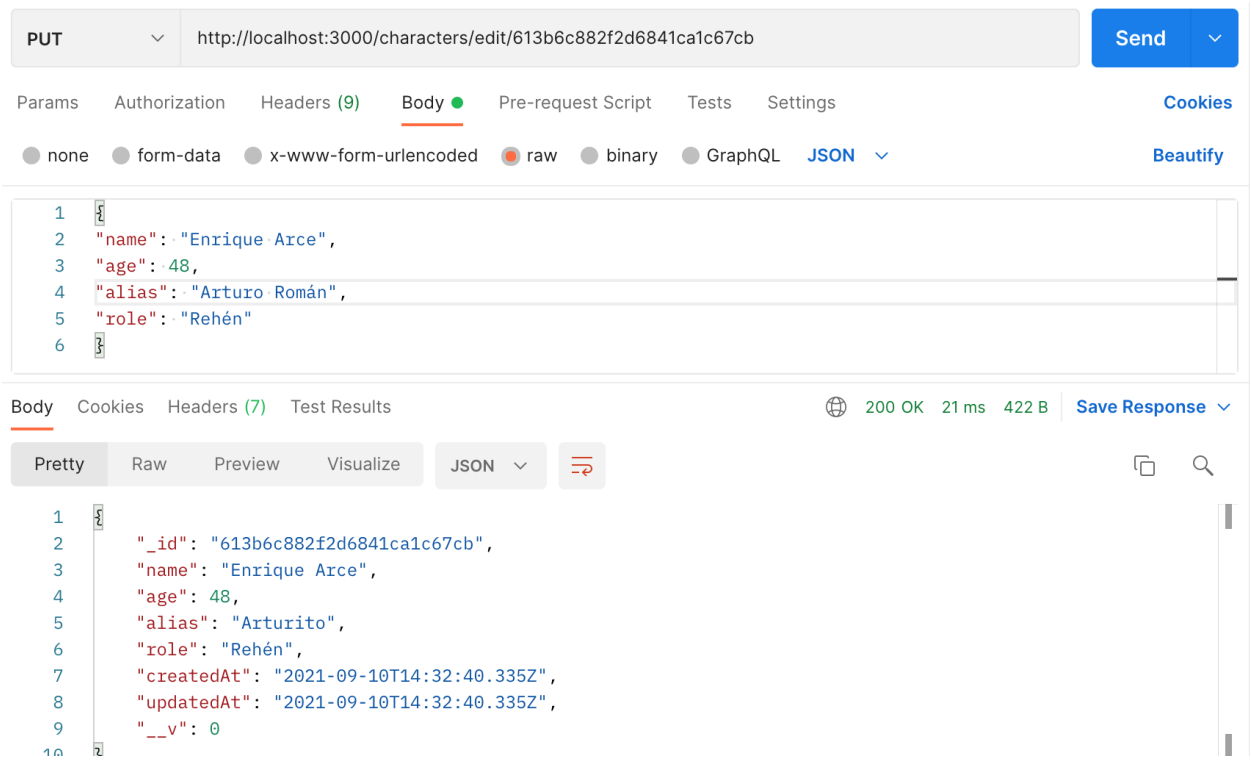
El método PUT es el más complejo de todos los que componen el CRUD, debido a que suele ser una "mezcla" de un delete y de un post. Necesitamos un id que recogeremos con req.params como hicimos en el delete y por otro lado tenemos que mandar información en el body, como hicimos en el post. Obligatoriamente se debe mandar el id en el body.

Como ya sabemos crear enpoints de tipo POST en nuestro servidor, vamos a editar un personaje dada su **id** utilizando los métodos PUT.

Para ello, crearemos un nuevo enpoint `/characters/edit/:id` en que enviaremos la id y los campos que queremos modificar.

```
router.put('/edit/:id', async (req, res, next) => {
    try {
        const { id } = req.params //Recuperamos el id de la url
        const characterModify = new Character(req.body) //instanciamos un nuevo Character con la información del body
        characterModify._id = id //añadimos la propiedad _id al personaje creado
        const characterUpdated = await Character.findByIdAndUpdate(id , characterModify)
        return res.status(200).json(characterUpdated)//Este personaje que devolvemos es el anterior a su modificación
    } catch (error) {
        return next(error)
    }
});
```

Cambiaremos el alias del personaje que acabamos de crear, es decir el alias "Arturito" lo vamos a modificar por "Arturo Román".



Para salir de dudas, puedes comprobar que exactamente se ha modificado el alias realizando de nuevo una petición get.

¡Hemos aprendido a crear una API con CRUD completa! 🎉