Server-side
Web Development

Unit 16. API REST with Node.js and Express

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2023-24

Index

1 API REST. A little reminder.

If you remember, we learned the foundations of REST API in unit 8. We took a look at the theory, and then we made some practical examples. Let's review the basic concepts:

- In a REST system, each resource is identified by a URI, optional parameters and a method.
- The method are GET, POST, PUT and DELETE, basically.

By identifying the resource to request (URI) and the command to apply to it, the server that offers this API REST provides a response to that request. This response is typically given by a message in JSON format.

We can check the services using a client application or using online tools like POSTMAN or hoopscotch as we saw in previous units.

2 Javascript and JSON

A JSON object, as you know, is defined by a series of properties and values. In Javascript we can create an object as a variable:

```
let person = {
      name: "Fidel",
      surname: "Oltra",
      age: 57
};
```

This object, translated to JSON format, looks like this:

```
{"name":"Fidel","surname":"Oltra","age":57}
```

Same for an array of objects:

```
let persons = [
  { name: "Fidel", surname: "Oltra", "age": 57},
  { name: "David", surname: "Sánchez", "age": 23},
  { name: "Martha", surname: "Lee", "age": 20},
  { name: "James", surname: "Williams", "age": 38}
];
```

Transformed into JSON format:

```
  [{"name": "Fidel", "surname": "Oltra", "age": 57},
  { "name": "David", "surname": "Sánchez", "age": 23},
  { "name": "Martha", "surname": "Lee", "age": 20},
  { "name": "James", "surname": "Williams", "age": 38}]
```

JavaScript offers some methods for converting data to JSON format and vice versa. These methods are `JSON.stringify` to convert a JavaScript object or array to JSON, and `JSON.parse` to convert a JSON string into a Javascript object.

```
let persons = [
  { name: "Fidel", surname: "Oltra", "age": 57},
  { name: "David", surname: "Sánchez", "age": 23},
  { name: "Martha", surname: "Lee", "age": 20},
  { name: "James", surname: "Williams", "age": 38}
];
// From array to JSON
let personsJSON = JSON.stringify(persons);
console.log(personsJSON);
// From JSON to array
let personsArray = JSON.parse(personsJSON);
console.log(personsArray);
```

The result:

```
[{"name":"Fidel","surname":"Oltra","age":57},{"name":"David","surname":"Sánchez","age":23},{"na

[
  { name: 'Fidel', surname: 'Oltra', age: 57 },
  { name: 'David', surname: 'Sánchez', age: 23 },
  { name: 'Martha', surname: 'Lee', age: 20 },
  { name: 'James', surname: 'Williams', age: 38 }
]
```

3 JSON and REST services

We will use JSON to create the response we send back when we receive a request. When preparing a response to a service using JSON format, this response usually will have a certain format. In general, in the responses we will use a structure based on:

- A boolean data (like ok) to indicate if the request was succesful or not
- An error message (like `error`) just in case something goes wrong (if ok is false)
- The response data (like `result` or any specific name) if the request has been succesful (if ok is true)
- Additionally we can add to the response an standard HTTP status code [HTTP status codes] (https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)

4 The EXPRESS framework

Express is a lightweight, flexible and powerful framework for developing web applications with Node. It is lightweight because it does not come by default with all the available functionality. Using Node and middleware modules, we can add later all the functionality required for each type of application. This way, we can the lighter version for simple web applications, and add more elements as we need them to develop more complex applications. That means we can use Express to create from static content servers (HTML, CSS and Javascript) to very complex web services.

More information about Express in the official Express website

4.1 Installing EXPRESS

Installing Express is as simple as with any other module that we want to incorporate into our Node project. We simply need to use npm in the project folder:

- first, if we don't have a `package.json` file yet, doing `npm init`
- then, once we have the `package.json` file, installing EXPRESS with `npm install express --save`

4.2 A simple example

Let's create a project named `ExpressTest`. Create the folder and install EXPRESS. Then, create an `index.js` file with this content:

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

We have created an EXPRESS object, and then we called a method `listen` to listen HTTP requests.

Now run the `index.js` script, go to you browser and write the URL `http://localhost:8080`.

We will get an error like this:

Cannot GET /

As the server it's not yet prepared to respond any request, we get an error message when trying to access any URL. Now, let's create some basic services.

5 Developing services

We must add some routes on our main server to support the services. Once we have initialized the application (app variable), we will just add the basic methods (get, post, put or delete), indicating for each one:

- the path or URI that should be attended
- the callback or function that will be executed

5.1 A simple service

Let's create a simple GET service attending the route `/welcome`. In your `index.js` file, add a method `app.get` like this:

```
const express = require('express');
let app = express();
app.get('/welcome', (req, res) => {
    res.send('Hi, welcome to my app');
  });
app.listen(8080);
```

If your server is running, restart it (close the terminal and run `node index.js` again). Then open your browser and go to the url `http://localhost:8080/welcome`. You should get the message:

Hi, welcome to my app

Let's analyze the code. The `app.get` method has two parameters:

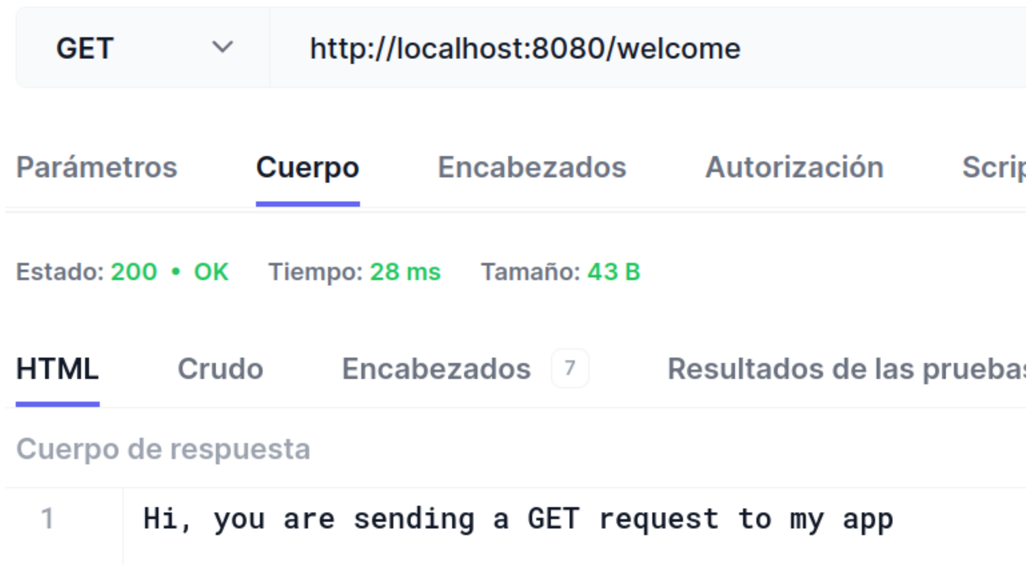
- the route, in this case `/welcome`
- the callback function, with two parameters: **req** and **res**.

req is the object with the client request, and **res** is the object with the response. We will use the **req** object to get information about the request, and **res** to send the response to the client. In our example, we are not reading any data from the request and we send (method `res.send`) a message as the only content in the response.

We can create the other services (post, put, delete) and return a different message in each of them.

```
const express = require('express');
let app = express();
app.get('/welcome', (req, res) => {
    res.send('Hi, you are sending a GET request to my app');
  });
app.post('/welcome', (req, res) => {
    res.send('Hi, you are sending a POST request to my app');
  });
app.put('/welcome', (req, res) => {
    res.send('Hi, you are sending a PUT request to my app');
  });
app.delete('/welcome', (req, res) => {
    res.send('Hi, you are sending a DELETE request to my app');
  });
app.listen(8080);
```

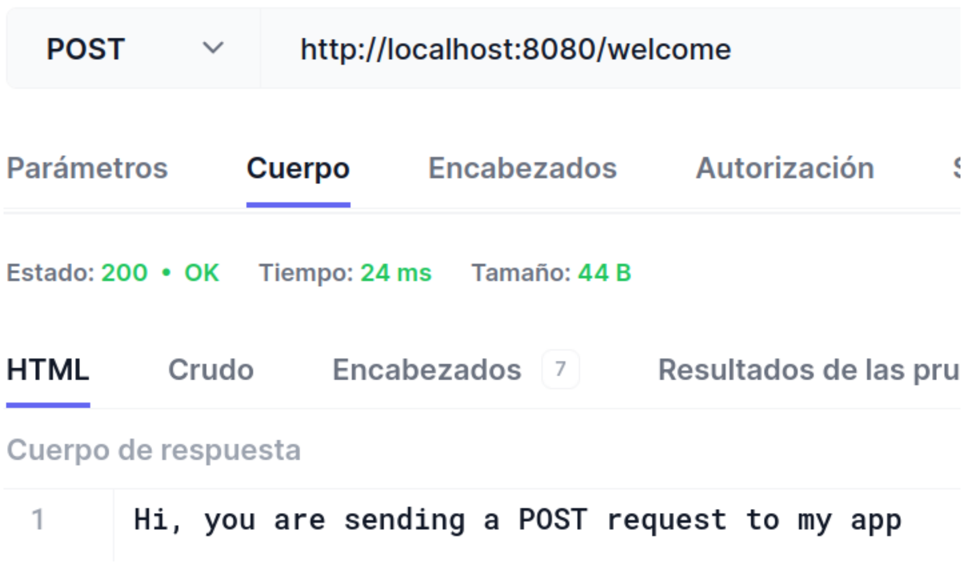
Now, to test the services, we must restart the app closing and executing again `index.js`. We only can send GET requests from the browser, so we need to use a web client like `Hoppscotch.io` or similar to test the other services.



```
Estado: 200 • OK    Tiempo: 28 ms    Tamaño: 43 B
```

```
Hi, you are sending a GET request to my app
```

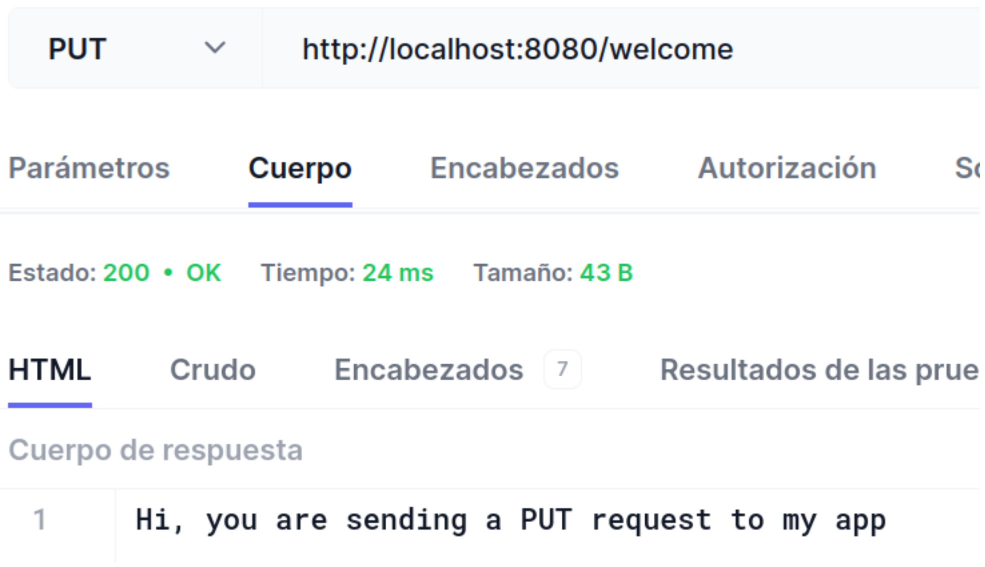
Figure 1: GET request



Estado: 200 • OK Tiempo: 24 ms Tamaño: 44 B

```
Hi, you are sending a POST request to my app
```

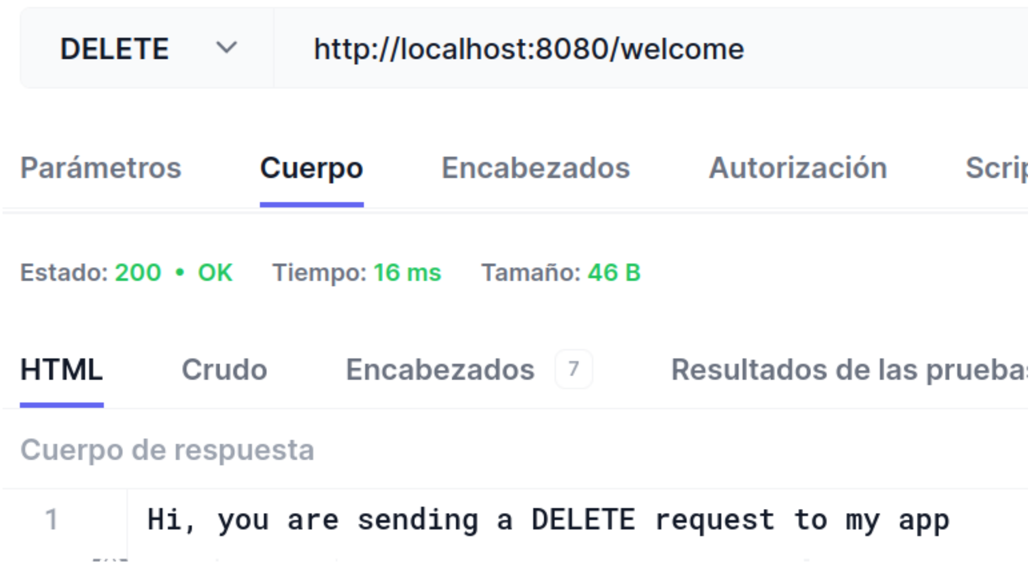
Figure 2: POST request



Estado: 200 • OK Tiempo: 24 ms Tamaño: 43 B

```
Hi, you are sending a PUT request to my app
```

Figure 3: PUT request



```
Hi, you are sending a DELETE request to my app
```

Figure 4: DELETE request

5.2 Exercise 01

Use the node library **os** to show the user name in the messages like we did in the previous unit.

6 Objects app, req and res

To create more complex services, we need to know more about the `app`, `req` and `res` objects.

6.1 The app object

The **application** is an instance of an **Express** object. In our example, we created a variable called `app`. This is and object with methods like:

- **use()** to add middleware to the project
- **set()** to create and assign a value to some project properties
- **get()** to get the value of some project properties
- **listen()** to specify which port the application will be listening on

The object has some more methods that we will be seeing in the next examples.

6.2 The req object

The **request** (in our example, **req**) object is created when a client sends a request to an Express server. This object contains several useful methods and properties that will allow us to access information contained in the petition, such as:

- **params** the parameters of the request
- **query** the parameters sent using the URL query string (after the ? in the URL)
- **body** the information sent using the body of the request
- **files** the files that a client uploads using a form
- **get()** to get some specific parameters from the request
- **path** to get the path or URL from the request
- **url** to get the URL and the `query string` parameters (those after the ? in the URL)

6.3 The `res` object

The **response** object (in our example, **`res`**) is created along with the **req** object. In the service code we will complete this object with some data that we want to send back to the client. The object has, among others, these methods and properties:

- **status(code)** to establish the state code for our response
- **set(header, value)** to establish a value to be returned in a header
- **redirect(status, url)** to redirect to another URL with a certain status code
- **send([status], body)** to return a content along with an optional status code
- **json([status], body)** to return JSON content along with an optional status code
- **render(view,[options])** to render a certain view as a response with the possibility of adding some options

7 A complete API REST

Let's create a complete API REST for our `contacts` collection in MongoDB. We will assume that we have a unique collection with the favourite restaurants and the pets as subdocuments inside the `contact` document. Thus, we will define the three schemas (for `contact`, `restaurant` and `pet`) in the same `contact.js` file.

contact.js

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

// Defining the scheme for contacts
// adding the restaurant and the pets as subdocuments
// inside the contact document

let restaurantSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 1,
    trim: true
  },
  address: {
    type: String,
    required: true,
    minlength: 1,
    trim: true
  },
  phone: {
    type: String,
    required: true,
    unique: true,
    trim: true,
    match: /^\d{9}$/
  }
});
let petSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 1,
    trim: true
  },
  type: {
    type: String,
    required: true,
    enum: ['dog', 'cat', 'others']
  }
});
let contactSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 1,
    trim: true
  },
  phone: {
    type: String,
    required: true,
    unique: true,
    trim: true,
    match: /^\d{9}$/
  },
  age: {
    type: Number,
    min: 18,
    max: 120
  },
  favouriteRestaurant: restaurantSchema,
  pets: [petSchema]
});

let Contact = mongoose.model('contact',contactSchema);
module.exports = Contact;
```

Now, in our `index.js` file, we must write:

index.js

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const Contact = require(__dirname + "/contact.js");

mongoose.connect('mongodb://172.17.0.1:27017/contacts_subdocuments');
```

This way, we are requiring some needed libraries and the **Contact** schema. Then we connect to a Mongo database called **contacts_subdocuments**

Now we can create the services.

7.1 POST

As our collection is empty, we need to add some contacts. First, we need to create the app and a middleware `express.json()` to parse the content of `req.body`:

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

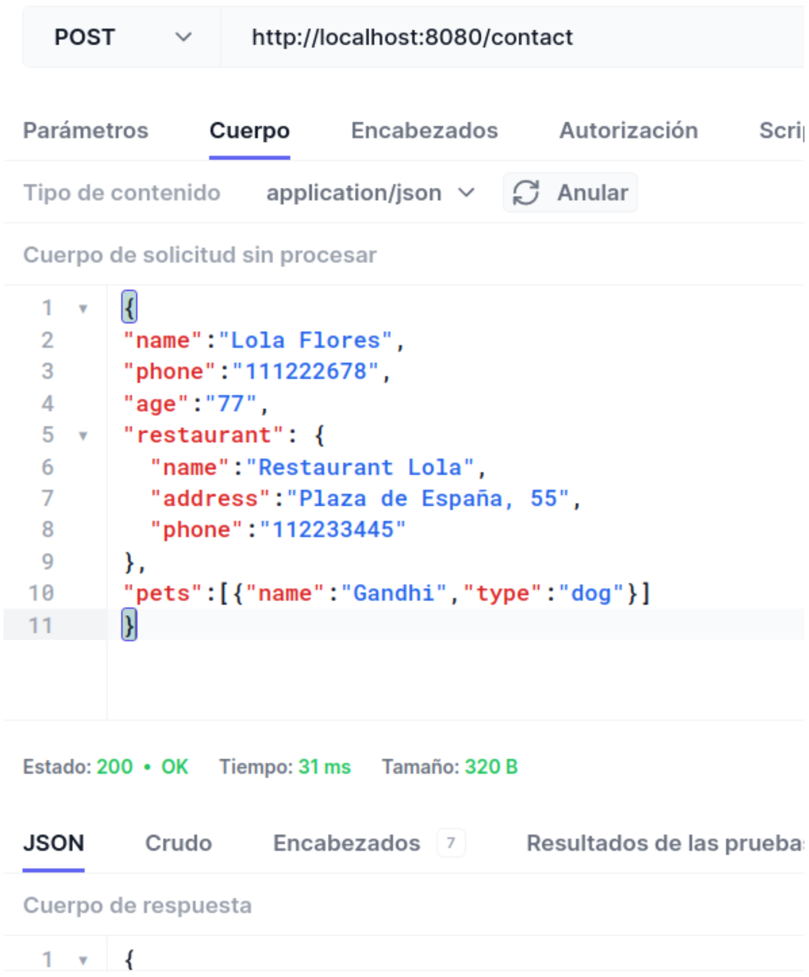
Now we can create the POST service:

```
app.post('/contact', (req, res) => {
    // we check if body has some content
    if(req.body) {
        // optionally, we can check if body includes all the information we
        ↪ need
        if(req.body.name && req.body.phone && req.body.age
            && req.body.restaurant && req.body.pets) {
            // we create the Contact
            let newContact = new Contact({
                name: req.body.name,
                phone: req.body.phone,
                age: req.body.age,
                favouriteRestaurant: req.body.restaurant,
                pets: req.body.pets
              });
            // and then we save the contact into the collection
            newContact.save().then(result => {
                res.status(200)
                .send({ok: true, resul: result});
              }).catch(error => {
                res.status(400)
                .send({ok: false,
                error: "Error adding the contact ",error,
                contact: newContact});
              });
            }
        else {
            // if some required parameter is missing in body
            res.status(400).send('Some parameter is missing');
        }
    }
    else {
        // if body is not valid
        res.status(400).send('Invalid request body');
    }
    });
```

Now, from our web client we must send a POST request with an `application/json` in the body with this format:

```
{
"name":"name",
"phone":"phone",
"age":"age",
"restaurant": {
   "name":"name of the restaurant",
   "address":"address of the restaurant",
   "phone":"phone of the restaurant"
},
"pets":[{"name":"name","type":"type"}, {"name":"name","type":"type"},...]
}
```

Let’s try with this one:


POST http://localhost:8080/contact

Parámetros Cuerpo Encabezados Autorización Scri

Tipo de contenido application/json Anular

Cuerpo de solicitud sin procesar

```
{
"name":"Lola Flores",
"phone":"111222678",
"age":"77",
"restaurant": {
  "name":"Restaurant Lola",
  "address":"Plaza de España, 55",
  "phone":"112233445"
},
"pets":[{"name":"Gandhi","type":"dog"}]
}
```

Estado: 200 • OK Tiempo: 31 ms Tamaño: 320 B

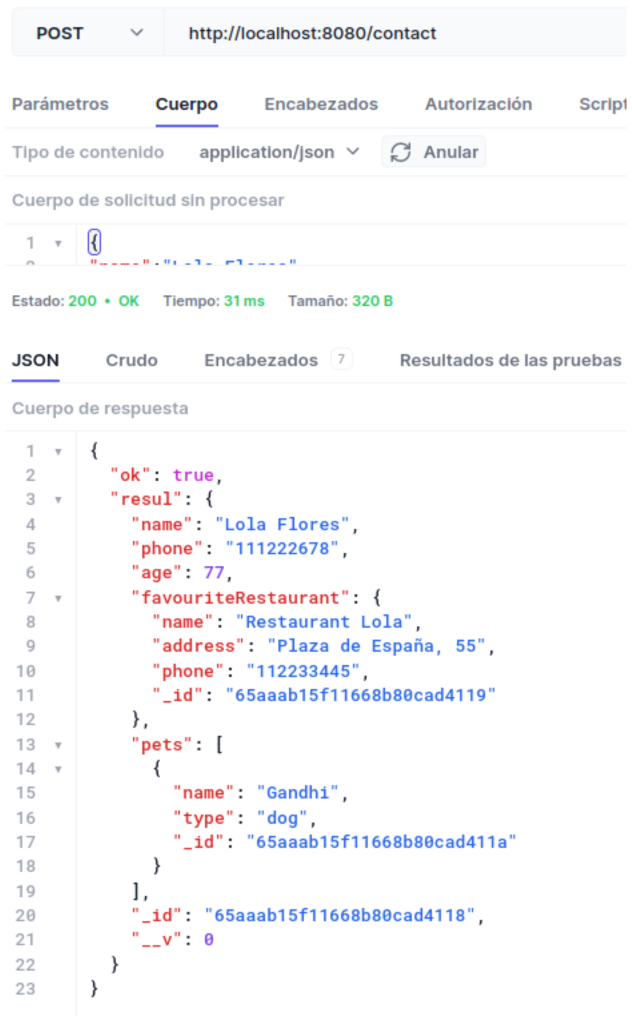
JSON Crudo Encabezados 7 Resultados de las prueba

Cuerpo de respuesta

```
{
```

Figure 5: a POST request

We will get this result:



```
{
  "ok": true,
  "resul": {
    "name": "Lola Flores",
    "phone": "111222678",
    "age": 77,
    "favouriteRestaurant": {
      "name": "Restaurant Lola",
      "address": "Plaza de España, 55",
      "phone": "112233445",
      "_id": "65aaab15f11668b80cad4119"
    },
    "pets": [
      {
        "name": "Gandhi",
        "type": "dog",
        "_id": "65aaab15f11668b80cad411a"
      }
    ],
    "_id": "65aaab15f11668b80cad4118",
    "__v": 0
  }
}
```

Figure 6: POST OK

> Remember to restart Node when you make changes to the code

We can simplify the code to relay on the model validation (remember that we only require the contact's name and phone):

```
app.post('/', (req, res) => {
  let newContact = new Contact({
      name: req.body.name,
      phone: req.body.phone,
      age: req.body.age,
      favouriteRestaurant: req.body.favouriteRestaurant,
      pets: req.body.pets
  });
  newContact.save().then(result => {
      res.status(200)
          .send({ok: true, result: result});
  }).catch(error => {
      res.status(400)
          .send({ok: false,
                error: "Error adding contact"});
  });
});
```

7.2 Exercise 02

- Try to do a new POST request to create a new contact but with the same phone number as the previous contact.
- Try to do a new POST request with a phone with more than 9 digits

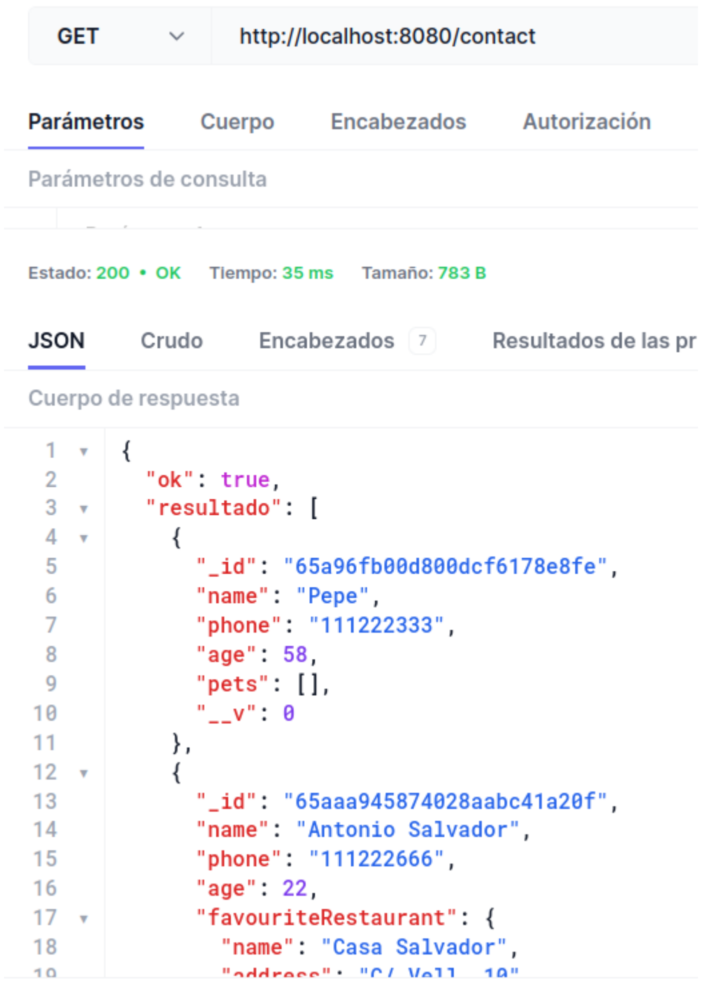
In both cases you should get an error. Why?

7.3 GET (all)

Of course, we can check if the contacts have been added to the collection directly from Mongo, but we also want to create a service that returns the contact list.

To get the whole list of contacts, we just need to do a `find` just as in Mongo and then create the response. As you can see, it's quite easy.

```
app.get('/contact', (req, res) => {
    Contact.find().then(result => {
      res.status(200)
        .send( {ok: true, result: result});
    }).catch (error => {
      res.status(500)
        .send( {ok: false,
              error: "Error reading contacts"});
    });
});
```



Estado: 200 • OK Tiempo: 35 ms Tamaño: 783 B

```
{
  "ok": true,
  "resultado": [
    {
      "_id": "65a96fb00d800dcf6178e8fe",
      "name": "Pepe",
      "phone": "111222333",
      "age": 58,
      "pets": [],
      "__v": 0
    },
    {
      "_id": "65aaa945874028aabc41a20f",
      "name": "Antonio Salvador",
      "phone": "111222666",
      "age": 22,
      "favouriteRestaurant": {
        "name": "Casa Salvador",
```

Figure 7: A GET request

7.4 GET by id

Usually when we want to get a document by its id, the id is sent as a parameter in the route. Thus, the method should be like this:

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

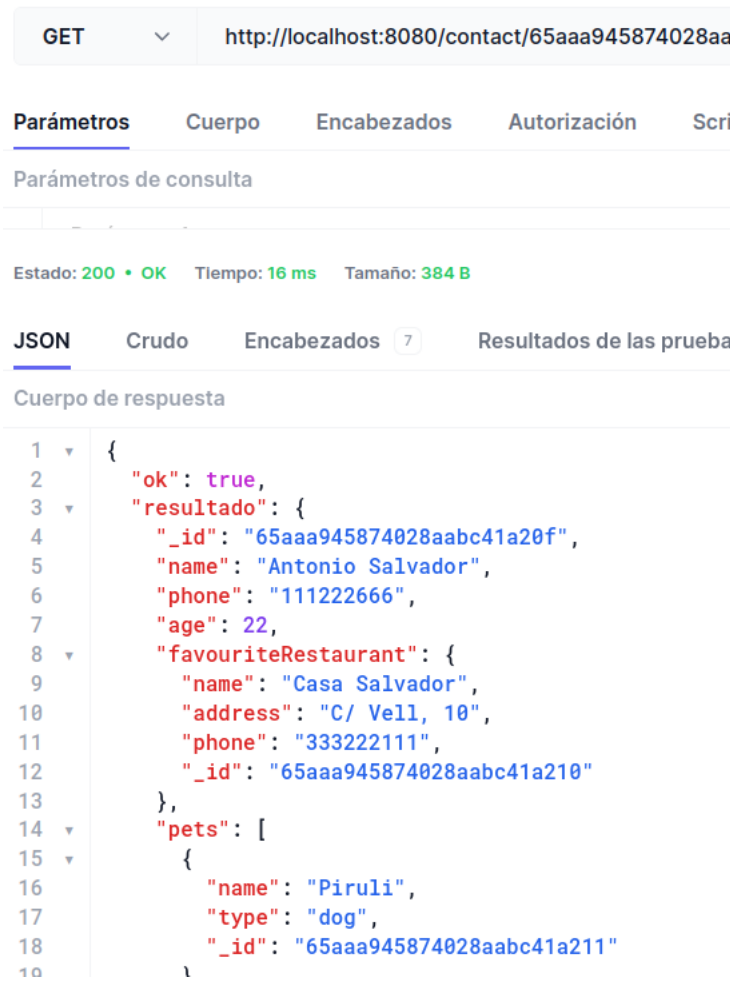
Inside the method, we can read the `:id` parameter by doing:

```
let idcontact = req.params.id;
```

And now, using the `findById` method we can find the contact with the given id.

```
app.get('/contact/:id', (req, res) => {
  let idContact = req.params.id;
  Contact.findById(idContact).then(result => {
    if(result)
      res.status(200)
        .send({ok: true, result: result});
    else
      res.status(400)
        .send({ok: false,
          error: "The contact doesn't exist"});
    }).catch (error => {
      res.status(400)
        .send({ok: false,
          error: "Error searching the contact"});
    });
});
```

Let's check the service from our web client:



http://localhost:8080/contact/65aaa945874028aa

Estado: 200 • OK Tiempo: 16 ms Tamaño: 384 B

```
{
  "ok": true,
  "resultado": {
    "_id": "65aaa945874028aabc41a20f",
    "name": "Antonio Salvador",
    "phone": "111222666",
    "age": 22,
    "favouriteRestaurant": {
      "name": "Casa Salvador",
      "address": "C/ Vell, 10",
      "phone": "333222111",
      "_id": "65aaa945874028aabc41a210"
    },
    "pets": [
      {
        "name": "Piruli",
        "type": "dog",
        "_id": "65aaa945874028aabc41a211"
```

Figure 8: A GET by the id

Try to seek an id that doesn't exist and see what happens.

> If we attach the parameter as a **query string** like in `/contact?id=...` we should read the parameter with `req.query.id` instead of `req.params.id`

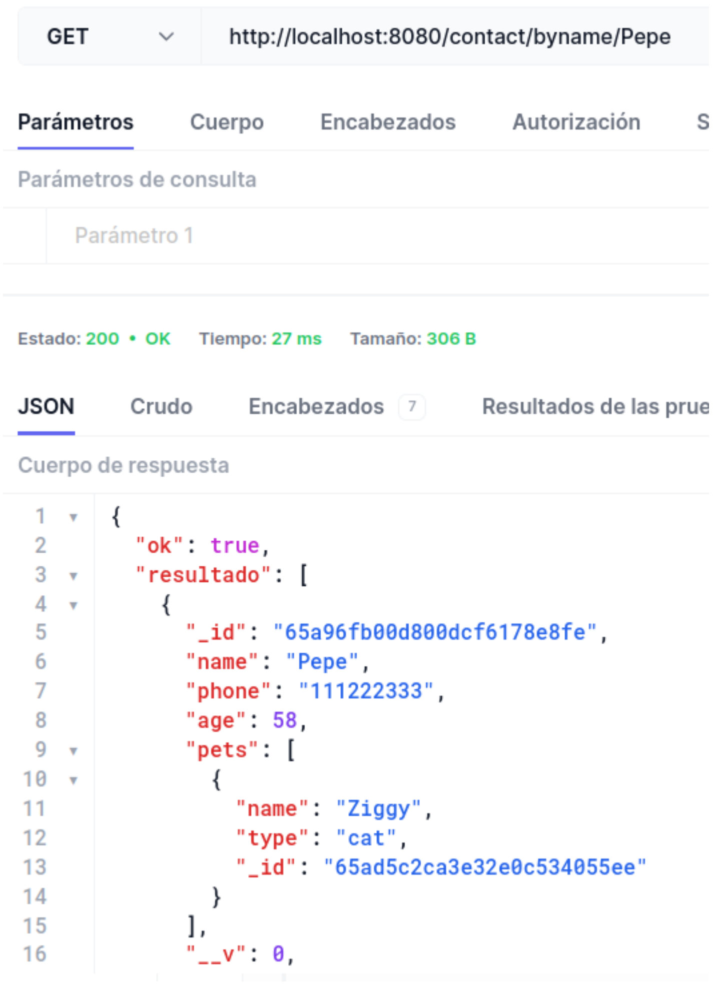
7.5 OTHER GETS

We can use the `find()` method to get the documents that match a condition. We can use the projections and the query operators

Let's create a service to find a contact by its name.

```
app.get('/contact/byname/:name', (req, res) => {
    let contactName = req.params.name;
    Contact.find({"name":contactName}).then(resultado => {
        if(resultado)
            res.status(200)
            .send({ok: true, resultado: resultado});
        else
            res.status(400)
            .send({ok: false,
            error: "The contact doesn't exist"});
        }).catch (error => {
            res.status(400)
            .send({ok: false,
            error: "Error searching the contact"});
        });

});
```


GET http://localhost:8080/contact/byname/Pepe

Estado: 200 • OK Tiempo: 27 ms Tamaño: 306 B

```
{
  "ok": true,
  "resultado": [
    {
      "_id": "65a96fb00d800dcf6178e8fe",
      "name": "Pepe",
      "phone": "111222333",
      "age": 58,
      "pets": [
        {
          "name": "Ziggy",
          "type": "cat",
          "_id": "65ad5c2ca3e32e0c534055ee"
        }
      ],
      "__v": 0,
```

Figure 9: A GET by the name

We can use regular expressions and `regex` to find the contacts which name matches a pattern. For example, if we try to use the previous method to find the contacts with the name “Lola” we will get none, because the complete name is “Lola Flores”. However, we can modify the service to make a partial search by the name using `regexp`.

```
let contactName = req.params.name;
var regex = new RegExp(contactName,'i');
Contact.find({"name": { $regex: regex}}).then(resultado => {
    ...
```

You can check that if we search for “Lola” or “Flores” we will get the same document, with the name “Lola Flores”, in both cases.

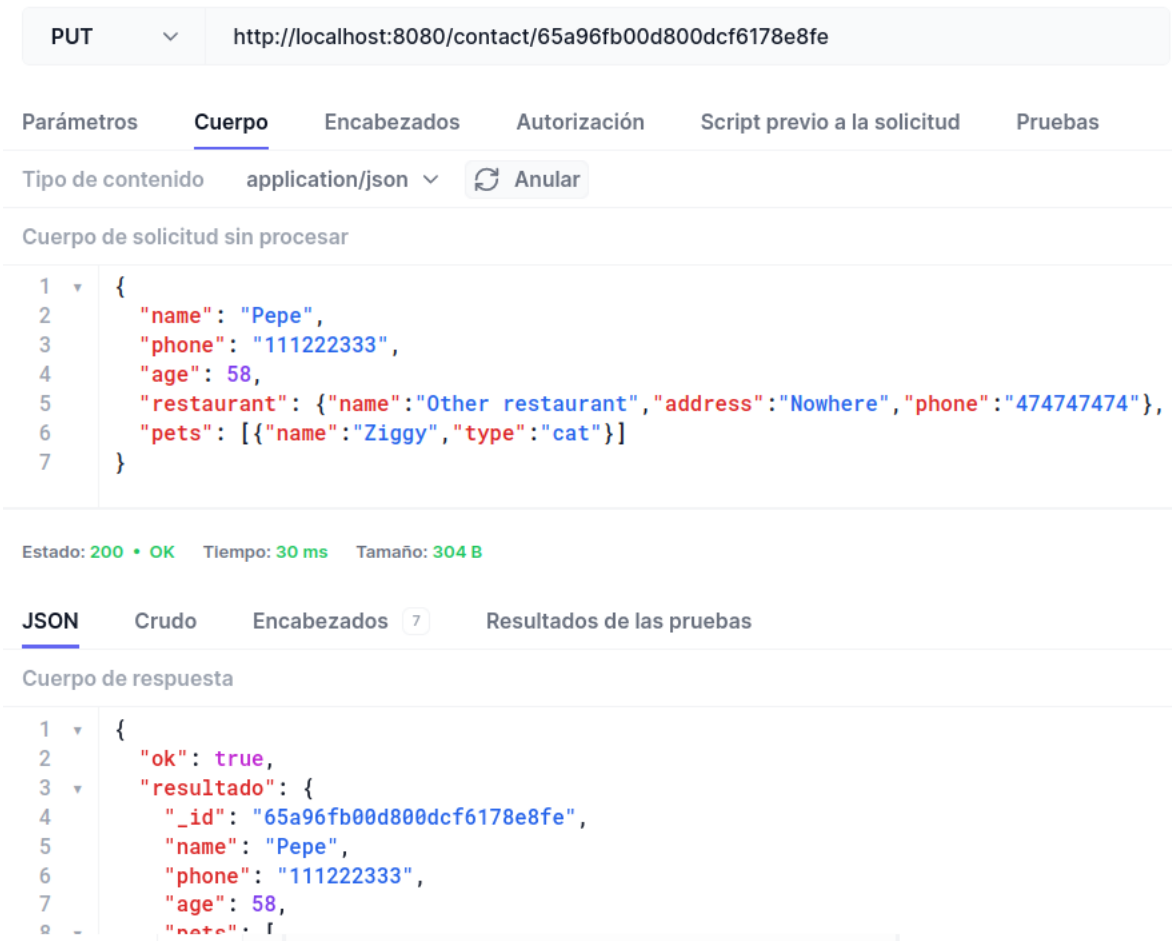
7.6 PUT

The **PUT** service is, in some way, a combination of **GET** and **POST**. As in the POST service, we will send in the body of the request the new data of the contact to be modified and the same middleware to get them. And, as in the GET by id service, we will pass as a parameter to the route the ID of the contact that we want to modify.

We can use the `findByIdAndUpdate` method to complete the whole task.

The code should be like this:

```
app.put('/contact/:id', (req, res) => {
  if(req.body) {
    if(req.body.name && req.body.phone && req.body.age
      && req.body.restaurant && req.body.pets) {
        Contact.findByIdAndUpdate(req.params.id, {
          $set: {
            name: req.body.name,
            phone: req.body.phone,
            age: req.body.age,
            favouriteRestaurant: req.body.restaurant,
            pets: req.body.pets
          }
          }, {new: true}).then(result => {
            res.status(200)
              .send({ok: true, result: result});
          }).catch(error => {
            res.status(400)
            .send({ok: false,
              error:"Error updating the contact"});
        });
    }
    else {
      res.status(400).send('Some parameter is missing');
    }
  }
  else {
    res.status(400).send('Invalid request body');
  }
});
```



```
http://localhost:8080/contact/65a96fb00d800dcf6178e8fe
application/json
```

```
{
  "name": "Pepe",
  "phone": "111222333",
  "age": 58,
  "restaurant": {"name":"Other restaurant","address":"Nowhere","phone":"474747474"},
  "pets": [{"name":"Ziggy","type":"cat"}]
}
```

Estado: 200 • OK Tiempo: 30 ms Tamaño: 304 B

```
{
  "ok": true,
  "resultado": {
    "_id": "65a96fb00d800dcf6178e8fe",
    "name": "Pepe",
    "phone": "111222333",
    "age": 58,
```

Figure 10: PUT using the id

We can achieve the same result relying on the model validation:

```
app.put('/:id', (req, res) => {
  Contact.findByIdAndUpdate(req.params.id, {
      $set: {
          name: req.body.name,
          phone: req.body.phone,
          age: req.body.age,
          favouriteRestaurant: req.body.favouriteRestaurant,
          pets: req.body.pets
      }
  }, {new: true, runValidators: true}).then(result => {
      if (result)
          res.status(200)
              .send({ok: true, result: result});
      else
          res.status(400)
              .send({ok: false, error: "Contact not found"});
  }).catch(error => {
      res.status(400)
          .send({ok: false,
                error:"Error updating contact: ", error});
  });
});
```

Note the use of `runValidators` to use the validation (see the previous unit).

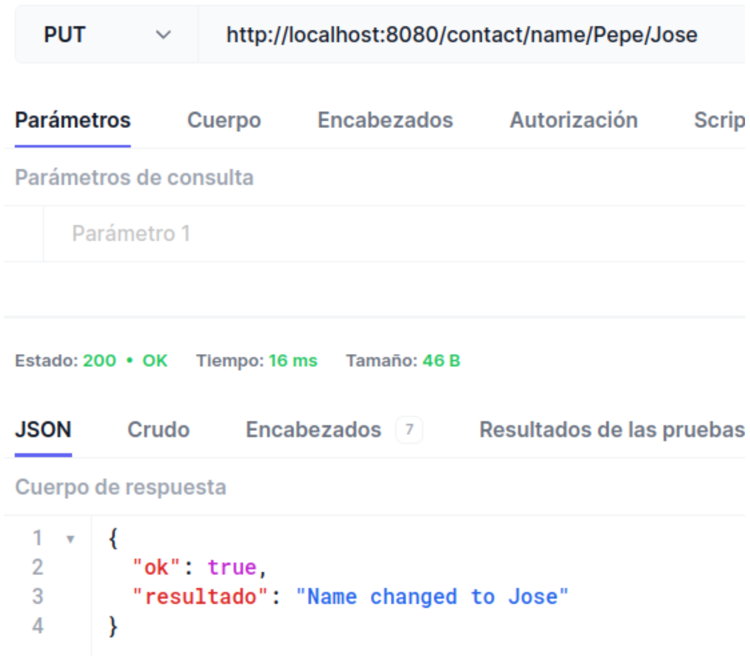
7.7 PUT using a filter

If we want to update a document matching a condition, we will use `findOneAndUpdate()`. To update all the documents matching a condition, we will use `updateMany()`.

How to use findOneAndUpdate() in Mongoose

An example: let's change the name of a contact. We will send both names (old and new) as parameters:

```
app.put('/contact/name/:name/:newname', (req, res) => {
    if(req.params.name && req.params.newname) {
        const filter = {name: req.params.name};
        const update = {name: req.params.newname}
        Contact.findOneAndUpdate(filter, update).then(result => {
                res.status(200)
                .send({ok: true, result: "Name changed to
                 ↪  "+req.params.newname});
            }).catch(error => {
                res.status(400)
                .send({ok: false,
                error:"Error updating the contact"});
            });
        }
    else {
        res.status(400).send('Some parameter is missing');
    }
});
```



```
PUT  http://localhost:8080/contact/name/Pepe/Jose
```

Estado: 200 • OK   Tiempo: 16 ms   Tamaño: 46 B

```
{
  "ok": true,
  "resultado": "Name changed to Jose"
}
```

Figure 11: PUT using a filter

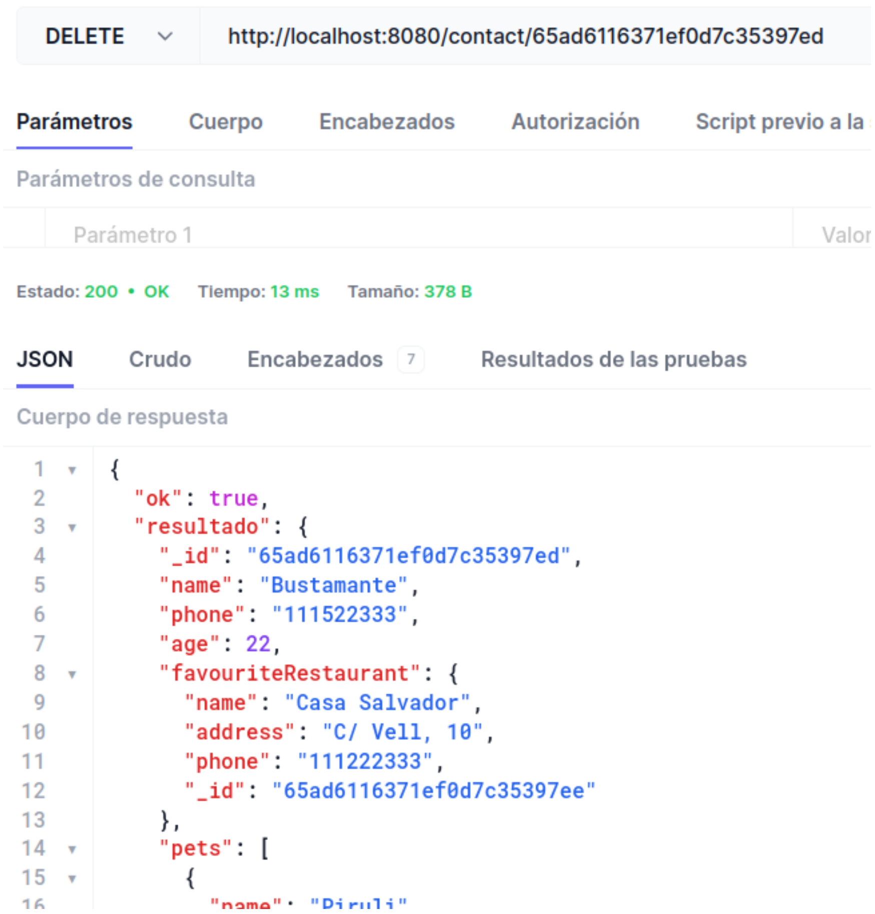
How to use updateMany() in Mongoose

7.8 DELETE a document

To delete contacts, we will use a URI similar to a contact get or update, but in this case associated with the DELETE command. We will pass as a parameter the ID of the contact to be deleted, and we will find and delete the contact with that id. We can use the method `findByIdAndDelete()`:

In this case, we are going to delete a contact using its id.

```
app.delete('/:id', (req, res) => {
    Contact.findByIdAndDelete(req.params.id).then(result => {
        if (result)
            res.status(200)
               .send({ok: true, result: result});
        else
            res.status(400)
               .send({ok: false, error: "Contact not found"});
    }).catch(error => {
        res.status(400)
           .send({ok: false,
                  error:"Error deleting contact"});
    });
});
```


DELETE http://localhost:8080/contact/65ad6116371ef0d7c35397ed

Estado: 200 • OK Tiempo: 13 ms Tamaño: 378 B

```
{
  "ok": true,
  "resultado": {
    "_id": "65ad6116371ef0d7c35397ed",
    "name": "Bustamante",
    "phone": "111522333",
    "age": 22,
    "favouriteRestaurant": {
      "name": "Casa Salvador",
      "address": "C/ Vell, 10",
      "phone": "111222333",
      "_id": "65ad6116371ef0d7c35397ee"
    },
    "pets": [
      {
        "name": "Piruli"
```

Figure 12: DELETE by id

We can use another condition as a filter, but using the `findOneAndDelete` method to delete. However, the method `findOneAndDelete` only removes the first document matching the filter.