ESCUELA SUPERIOR DE INGENIERÍA

Nombre de Asignatura

Titulación

PNET Hito 2 y 3

Autores:

Claudia Silvestre Muñoz y Jesús Márquez Delgado

Índice General

1. Creación de la API RESTful

En la carpeta donde se localiza nuestro proyecto abriremos la consola e introduciremos los siguientes comandos.

1.1. npm init

Para crear el package.json de nuestro proyecto escribiremos el comando:

npm init

y rellenaremos los campos como el nombre del package, la descripción y el nombre del autor/autores.

```
D:\Pácticas\PNET-Project\Hito 2>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (hito-2) hito 2 y 3
Sorry, name can only contain URL-friendly characters.
package name: (hito-2) hito2y3
version: (1.0.0)
description: api reservas con mongodb
entry point: (index.js)
test command:
git repository:
keywords:
author: Claudia Silvestre y Jesus Marquez
license: (ISC)
About to write to D:\Pácticas\PNET-Project\Hito 2\package.json:

{
  "name": "hito2y3",
  "version": "1.0.0",
  "description": "api reservas con mongodb",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Claudia Silvestre y Jesus Marquez",
  "license": "ISC"
}
```

```
{
  "name": "hito2y3",
  "version": "1.0.0",
  "description": "api reservas con mongodb",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Claudia Silvestre y Jesus Marquez",
  "license": "ISC"
}
```

1.2. Instalación de dependencias

Ahora pasaremos a instalar las dependencias necesarias para la ejecución de nuestra API, en este caso estas son “express, morgan, body-parser y cors”, para ello introducimos el comando:

npm install express morgan body-parser mongodb cors --save

con la opción --save para que se almacene en nuestro package.json.

```
D:\Pácticas\PNET-Project\Hito 2>npm install express morgan body-parser mongodb cors --save
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN hito2y3@1.0.0 No repository field.

+ cors@2.8.5
+ morgan@1.10.0
+ body-parser@1.19.0
+ mongodb@3.6.6
+ express@4.17.1
added 70 packages from 46 contributors and audited 70 packages in 4.085s
found 0 vulnerabilities
```

```
"dependencies": {
  "body-parser": "^1.19.0",
  "cors": "^2.8.5",
  "express": "^4.17.1",
  "mongodb": "^3.6.6",
  "morgan": "^1.10.0"
}
```

1.3. Creación del index.js

Creamos un fichero llamado "**index.js**" en la raíz del proyecto.

Añadimos las constantes necesarias:

```
const express = require('express');
const app = express();
const logger = require('morgan');
const http = require('http');
const path = require('path');
const PORT = process.env.PORT || 8080;
const bodyParser = require('body-parser');
const baseAPI = '/api/v1';
const cors = require('cors');
```

Añadimos configuraciones para la app:

```
app.use(bodyParser.json());
app.use(logger('dev'));
app.use(bodyParser.urlencoded({
    extended: true
}));

app.use(cors());
```

1.4. Routers

Para mantener los recursos organizados y empaquetados dentro de la API utilizamos *routes.*

Creamos la carpeta "**routes**" en la raíz del proyecto y creamos el fichero "**bookings.js**" dentro de "routes".

Añadimos en el fichero **bookings.js** las siguientes líneas:

```
'use strict';

const express = require('express');
const router = express.Router();
```

E implementamos las operaciones definidas para "bookings" en este fichero:

```
//Recuperar todas las reservas.
router.get('/', function (req, res) {
    bookingsService.getAll((err, bookings) => {
            if (err) {
                res.status(500).send({
                    msg: err
                });
            } else if (bookings === null){
                res.status(500).send({
                    msg: "bookings null"
                });
            } else {
                res.status(200).send(bookings);
            }
        }
    );
});

//Recuperar una única reserva existente por ID.
router.get('/:_id', function (req, res) {
    let _id = req.params._id;
    bookingsService.get(_id, (err, booking) => {
            if (err) {
                res.status(500).send({
                    msg: err
                });
            } else if (booking === null){
                res.status(500).send({
                    msg: "bookings null"
                });
            } else {
                res.status(200).send(booking);
            }
        }
    );
});
```

```
//Añadir una nueva reserva.
router.post('/', function (req, res) {
    let booking = req.body;
    bookingsService.add(booking, (err, booking) => {
            if (err) {
                res.status(500).send({
                    msg: err
                });
            } else if (booking !== null) {
                res.send({
                    msg: 'Booking created!'
                });
            }
        }
    );
});

//Actualizar una reserva existente por ID.
router.put('/:_id', function (req, res) {
    const _id = req.params._id;
    const updatedBooking = req.body;
    bookingsService.update(_id, updatedBooking, (err, numUpdates) => {
        if (err || numUpdates === 0) {
            res.status(500).send({
                msg: err
            });
        } else {
            res.status(200).send({
                msg: 'Booking updated!'
            });
        }
    });
});
```

```
//Eliminar todas las reservas.
router.delete('/', function (req, res) {
    bookingsService.removeAll((err) => {
        if (err) {
            res.status(500).send({
                msg: err
            });
        } else {
            res.status(200).send({
                msg: 'Bookings deleted!'
            });
        }
    });
});

//Eliminar una única reserva existente por ID.
router.delete('/:_id', function (req, res) {
    let _id = req.params._id;
    bookingsService.remove(_id, (err) => {
        if (err) {
            res.status(500).send({
                msg: err
            });
        } else {
            res.status(200).send({
                msg: 'Booking deleted!'
            });
        }
    });
});
```

Añadimos la siguiente línea al final del fichero **bookings.js** para exportar el enrutador que hemos creado:

```
module.exports = router;
```

Para importar el enrutador y utilizarlo en nuestra API escribimos las siguientes líneas en el **index.js**:

```
const bookings = require('./routes/bookings');
```

```
app.use('/bookings', bookings);
```

1.5. Persistencia

A continuación, añadimos persistencia a nuestra API utilizando la base de datos MongoDB Atlas.

En primer lugar, creamos el fichero "**bookings-service.js**" dentro de la carpeta routes y añadimos las siguientes constantes:

```
'use strict';

const MongoClient = require('mongodb').MongoClient;
let db;
let ObjectId = require('mongodb').ObjectID;
const Bookings = function () {
};
```

Añadimos los parámetros necesarios para conectarnos a la base de datos:

```
Bookings.prototype.connectDb = function (callback) {
    MongoClient.connect("mongodb+srv://testPNET:testPNET123@jmd-pnet-2020-2021.tejai.mongodb.net/myFirstDatabase?retryWrites=true&w=majority",
        {useNewUrlParser: true, useUnifiedTopology: true},
        function (err, database) {
            if (err) {
                callback(err);
            }

            db = database.db('jmd-pnet-2020-2021').collection('bookings');

            callback(err, database);
        });
};
```

Y los métodos para trabajar con la base de datos:

```
Bookings.prototype.add = function (booking, callback) {
    return db.insertOne(booking, callback);
};

Bookings.prototype.get = function (_id, callback) {
    return db.find({_id: ObjectId(_id)}).toArray(callback);
};

Bookings.prototype.getAll = function (callback) {
    return db.find({}).toArray(callback);
};

Bookings.prototype.update = function (_id, updatedBooking, callback) {
    delete updatedBooking._id;
    return db.updateOne({_id: ObjectId(_id)}, {$set: updatedBooking}, callback);};

Bookings.prototype.remove = function (_id, callback) {
    return db.deleteOne({_id: ObjectId(_id)}, callback);
};

Bookings.prototype.removeAll = function (callback) {
    return db.deleteMany({}, callback);
};
```

En **bookings.js** importamos el fichero bookings-service.js:

```
const express = require('express');
const router = express.Router();
const bookingsService = require('./bookings-service');
```

Editamos el fichero **index.js** e importamos el fichero bookings-service.js justo antes de la importación del fichero bookings.js:

```
const bookingsService = require('./routes/bookings-service');
const bookings = require('./routes/bookings');
```

Finalmente, en **index.js** instanciamos el servidor y lo ponemos para escuchar en el puerto PORT (8080 por defecto). Antes de arrancar el servidor, tenemos que conectar con la BBDD.

```
const server = http.createServer(app);

bookingsService.connectDb(function (err) {
    if (err) {
        console.log('Could not connect with MongoDB – bookingsService');
        process.exit(1);
    }

    server.listen(PORT, function () {
        console.log('Server up and running on localhost:' + PORT);
    });
});
```

1.6. Inicio de la API

Para iniciar la API REST vamos a la consola de comandos y en la raíz de nuestro proyecto ejecutamos el siguiente comando:

node index.js

Si todo ha ido bien nos aparecerá en la consola de comandos:

```
Server up and running on localhost:8080
```

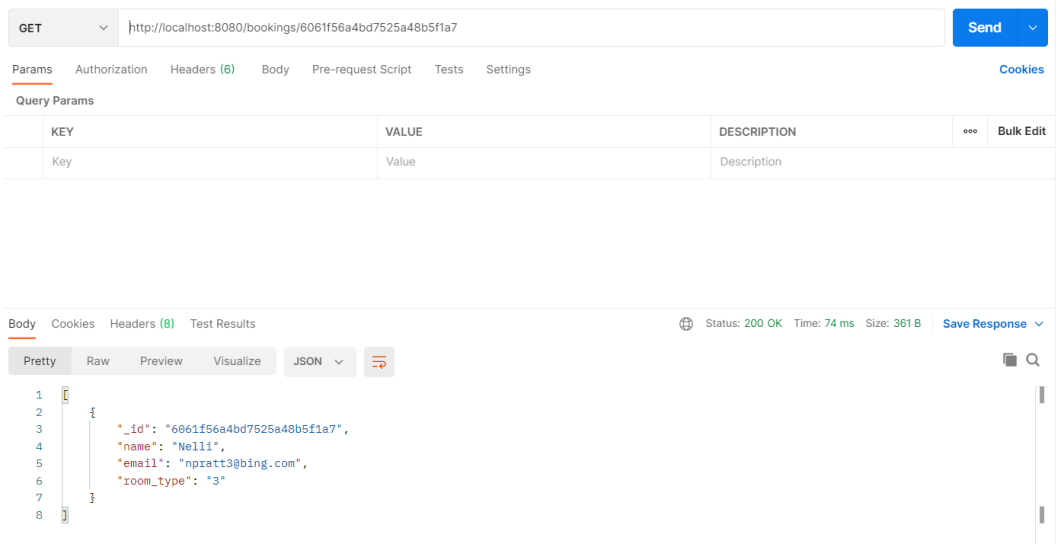
2. Servicios de la API mediante POSTMAN

2.1. GET

En POSTMAN configuramos un envío de peticiones GET y le damos a "SEND".

Si queremos consultar la reserva con ID: 6061f56a4bd7525a48b5f1a7

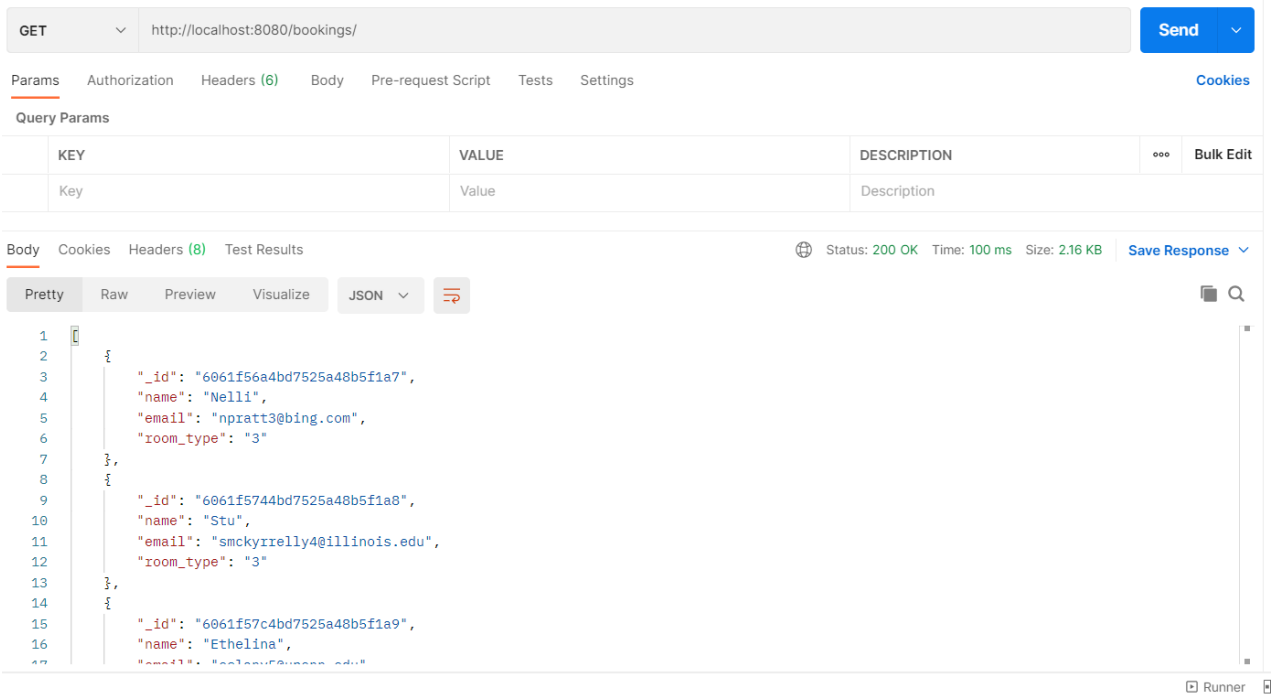
- Método: GET
- URL: http://localhost:8080/bookings/6061f56a4bd7525a48b5f1a7



2.2. GET All

En POSTMAN configuramos un envío de peticiones GET y le damos a "SEND".

- Método: GET
- URL: http://localhost:8080/bookings/

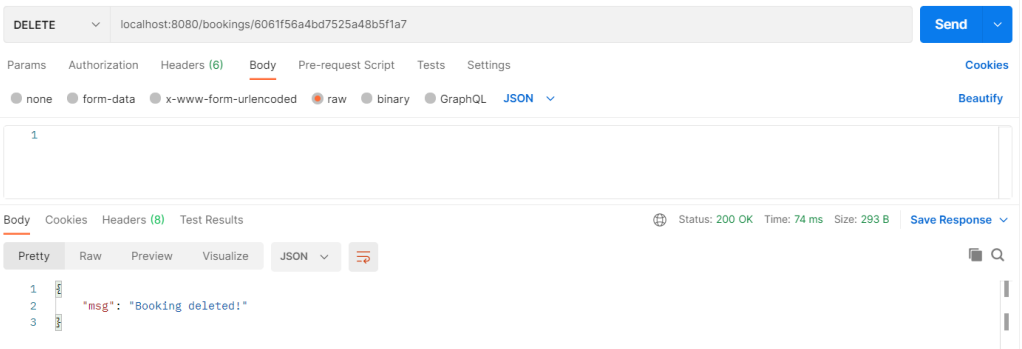


2.3. DELETE

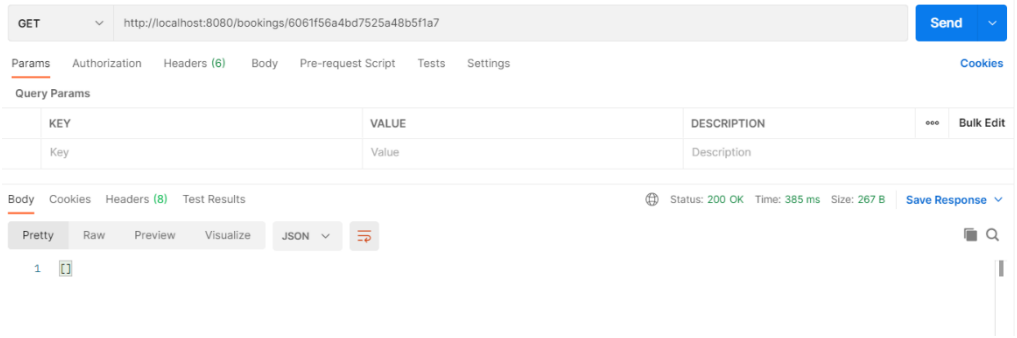
En POSTMAN configuramos un envío de peticiones DELETE y le damos a "SEND".

Si queremos borrar la reserva con ID: 6061f56a4bd7525a48b5f1a7

- Método: DELETE
- URL: http://localhost:8080/bookings/6061f56a4bd7525a48b5f1a7



Para comprobarlo usamos el GET para esa ID de nuevo:

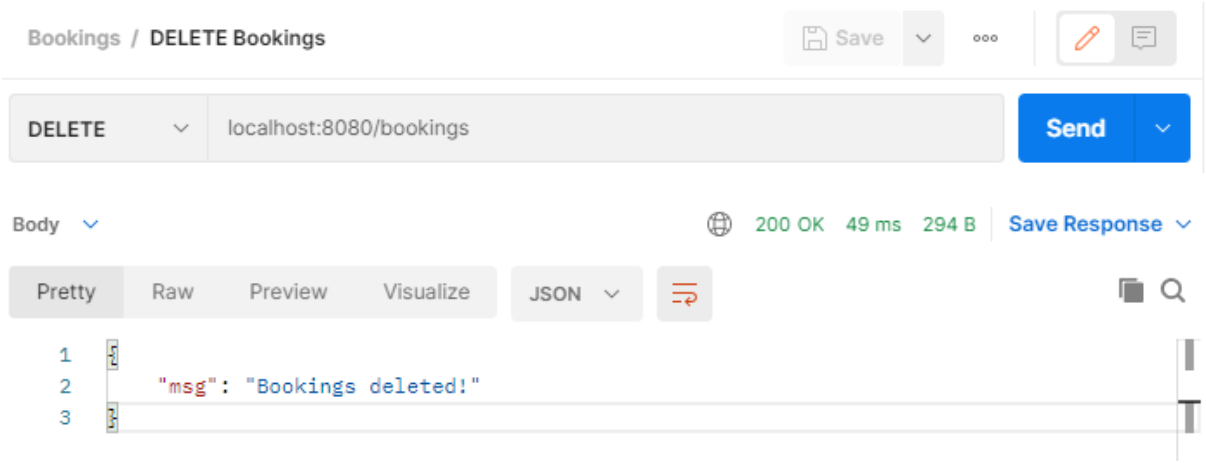


Vemos que la reserva se ha eliminado correctamente.

2.4. DELETE All

En POSTMAN configuramos un envio de peticiones DELETE y le damos a “SEND”

- Método: DELETE
- URL: http://localhost:8080/bookings/

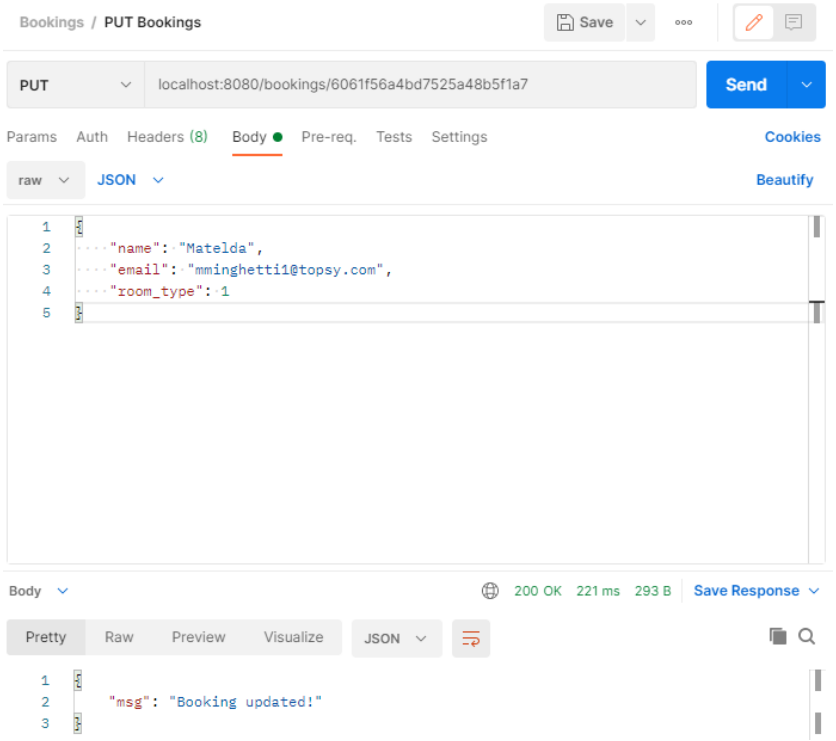


2.5. PUT

En POSTMAN configuramos un envio de peticiones PUT y le damos a “SEND”

Si queremos modificar la reserva con ID: 6061f56a4bd7525a48b5f1a7

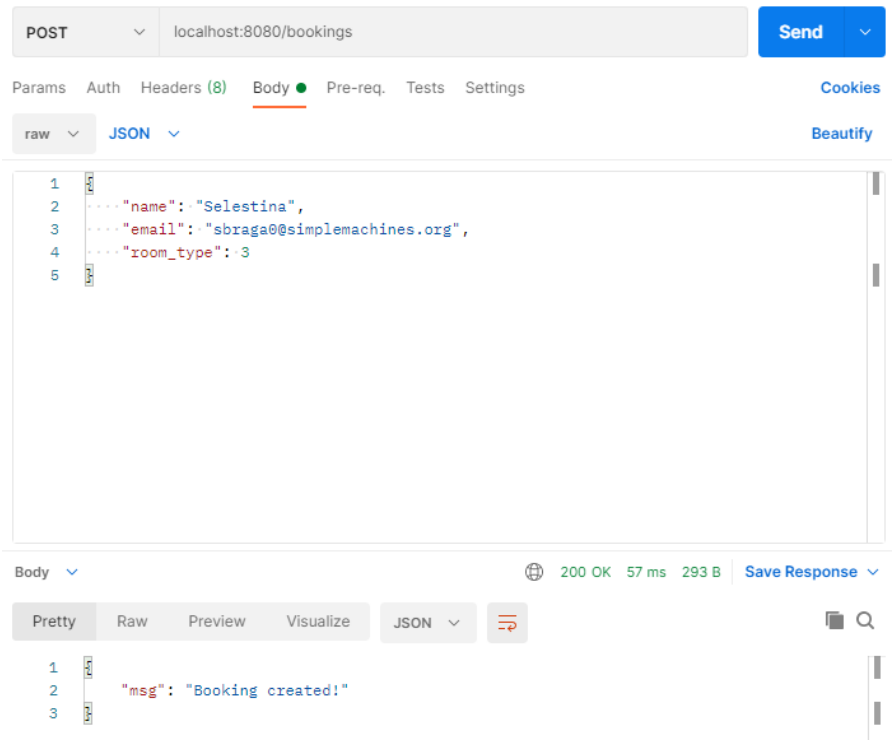
- Método: PUT
- URL: http://localhost:8080/bookings/6061f56a4bd7525a48b5f1a7
- JSON: Con los datos actualizados



2.6. POST

En POSTMAN configuramos un envio de peticiones POST y le damos a “SEND”

- Método: POST
- URL: http://localhost:8080/bookings/6061f56a4bd7525a48b5f1a7
- JSON: Con los datos a guardar



3. Invocar métodos de la API desde funciones JavaScript

En un fichero llamado "**wsinvocations.js**" implementamos todos los métodos necesarios para realizar las 6 operaciones y en otro llamado "**index.html**" se encuentra la estructura de una página web que creamos.

3.1. GET ALL

En esta función utilizamos la función AJAX con las opciones:

- type: GET
- url: "http://localhost:8080/bookings/" (uri del recurso al que queremos acceder, en nuestro caso, la url para las reservas)
- dataType: el tipo de dato que esperamos recibir tras la invocación, en nuestro caso "json".

En caso de ser válidos estos parámetros, las reservas se mostrarán en la parte del index.html correspondiente al atributo resBooking. En caso contrario, se mostrará un mensaje de error.

```
//Obtiene y muestra todas las reservas en pantalla
function getAllBookings() {
    var myUrl = "http://localhost:8080/bookings/";
    $.ajax({
        type: "GET",
        dataType: "json",
        url: myUrl,
        success: function(data) {
            $("#resBooking").html(jsonsToHtml(data));
            insertAllOnClick(data);
        },
        error: function(res) {
            alert("ERROR " + res.statusText);
        }
    });
}
```

3.2. GET

En esta función recibimos el ID de una reserva y utilizamos la función AJAX con las opciones:

- type: GET
- url: "http://localhost:8080/bookings/" + bookingId (uri del recurso al que queremos acceder, en nuestro caso, la url para las reservas con el ID de la reserva pasado por parámetro)
- dataType: el tipo de dato que esperamos recibir tras la invocación, en nuestro caso "json".

En caso de ser válidos estos parámetros, la reserva se mostrará en la parte del index.html correspondiente al atributo resBooking. En caso contrario, se mostrará un mensaje de error.

```
//Recibe el ID de una reserva y la busca para mostrarla en pantalla
function getBooking(bookingId) {
    var myUrl = "http://localhost:8080/bookings/" + bookingId;
    $.ajax({
        type: "GET",
        dataType: "json",
        url: myUrl,
        success: function(data) {
            $("#resBooking").html(jsonToHtml(data[0]));
            insertOnClick(data[0])
        },
        error: function(res) {
            alert("ERROR:" + res.statusText);
        }
    });
}
```

3.3. DELETE

En esta función recibimos el ID de una reserva y utilizamos la función AJAX con las opciones:

- type: DELETE
- url: "http://localhost:8080/bookings/" + bookingId (uri del recurso al que queremos acceder, en nuestro caso, la url para las reservas con el ID de la reserva pasado por parámetro para eliminar esa reserva)
- dataType: el tipo de dato que esperamos recibir tras la invocación, en nuestro caso "text".

En caso de ser válidos estos parámetros, se llama a la parte del index.html correspondiente al atributo resBooking. En caso contrario, se mostrará un mensaje de error.

```
//Recibe un ID de una reserva y si existe, lo borra de la base de datos
function deleteBooking(bookingId) {
    var myUrl = "http://localhost:8080/bookings/" + bookingId;
    $.ajax({
        type: "DELETE",
        dataType: "text",
        url: myUrl,
        success: function (data) {
            $("#resBooking").html(data);
        },
        error: function (res) {
            alert("ERROR " + res.statusText);
        }
    });
}
```

3.4. DELETE All

3.5. PUT

En esta función recibimos el ID de una reserva y utilizamos la función AJAX con las opciones:

- type: PUT
- url: "http://localhost:8080/bookings/" + booking._id (uri del recurso al que queremos acceder, en nuestro caso, la url para las reservas con el ID de la reserva pasado por parámetro para modificar esa reserva)
- contentType: el tipo de dato que vamos a enviar, en este caso “JSON”.
- dataType: el tipo de dato que esperamos recibir tras la invocación, en nuestro caso "text".
- data: JSON con los datos actualizados

Este método va asignado a un botón que acompaña a los datos recibidos de cada get y si se pulsa el botón y se encuentra la reserve a actualizar, esta se actualiza en la base de datos y en la página aparece un mensaje de acierto, en caso contrario se mostrará un mensaje de error.

```
//Recibe una reserva y actualiza sus valores con los introducidos en sus campos de entrada
function putBooking(booking) {
    var myUrl = "http://localhost:8080/bookings/" + booking._id;
    $.ajax({
        type: "PUT",
        url: myUrl,
        contentType: "application/json",
        dataType: "text",
        data: JSON.stringify(getBookingInputValues(booking)),
        success: function (data) {
            $("#resBooking").html(data);
        },
        error: function (res) {
            alert("ERROR: " + res.statusText);
        }
    });
}
```

3.6. POST

En esta función miramos los datos introducidos en los campos del POST y se publican como una nueva reserva en la base de datos:

- type: POST
- url: "http://localhost:8080/bookings/"
- contentType: el tipo de dato que vamos a enviar, en este caso “JSON”.
- dataType: el tipo de dato que esperamos recibir tras la invocación, en nuestro caso "text".
- data: JSON con los datos de la reserva

Este método va asignado a un botón que acompaña a los campos a rellenar de la reserva, una vez pulsado si la petición se recibe con éxito mostrará un mensaje de acierto, en caso contrario se mostrará un mensaje de error.

```
//Publica una nueva reserva a partir de los datos introducidos en las entradas
function postBooking() {
    $.ajax({
        type: "POST",
        url: "http://localhost:8080/bookings/",
        contentType: "application/json",
        dataType: "text",
        data: JSON.stringify(getPostBookingInputValues()),
        success: function(data) {
            $("#resBooking").html(data);
        },
        error: function(res) {
            alert("ERROR: " + res.statusText);
        }
    });
}
```

4. Consumir operaciones desde una página web

4.1. jsonsToHtml

Al enviar la petición, recibimos un JSON con los datos, estos datos son tratados dentro de un método el cual extrae cada elemento de la lista de JSON y los pasa por otro método llamado “jsonToHtml”.

```
// Recibe una lista de reservas y devuelve sus respectivos html en forma de cadena
function jsonsToHtml(bookings) {
    let string = "";
    for (var i = 0; i < Object.keys(bookings).length; ++i)
        string += jsonToHtml(bookings[i]);
    return string;
}
```

El método “jsonToHtml” crea un string para luego ser interpretado por él .html(), este se compone de un div con la clase “admin_div” el cual pondrá un estilo con el borde negro para dividir cada elemento, luego cada atributo estará dentro de un párrafo.

Excepto el id, cada atributo tendrá un label y un input para poder ser modificados posteriormente.

Por último, debajo del todo se insertarán dos botones, uno para hacer el PUT y otro para el DELETE.

```
// Recibe una reserva en forma de json y devuelve el html
// en forma de cadena equivalente junto con sus botones
function jsonToHtml(booking) {
    let string = "<div class='admin_div' >";
    string += "<p>" + Object.keys(booking)[0] + ": "
     + booking[Object.keys(booking)[0]] + "</p>"
    for(var i = 1; i < Object.keys(booking).length; ++i)
    {
        string += "<p><label for='" + booking._id + Object.keys(booking)[i] + "'>"
            + Object.keys(booking)[i] + ": </label>"
            + "<input type='text' id='" + booking._id + Object.keys(booking)[i] + "' value='"
            + booking[Object.keys(booking)[i]] + "'></p>";
    }
    string += '<input type="button" value="Update Booking" id="' + booking._id + 'put" />'
    string += '<input type="button" value="Delete Booking" id="' + booking._id + 'delete" />'
    string += "</div>"
    return string;
}
```

4.2. insertAllOnClick

Este método se ejecutará después de crear el html para mostrar los datos, este recibe todas las reservas e inserta en los botones de cada reserva los métodos onclick para el “PUT” y el “DELETE”

```
// Asigna los eventos onclick de los botones de un conjunto de reservas
function insertAllOnClick(bookings) { bookings.forEach(booking => {insertOnClick(booking);}); }
```

```
// Asigna los eventos onclick de los botones de una reserva
function insertOnClick(booking) {
    $("#" + booking._id + "put").click(function () { putBooking(booking) });
    $("#" + booking._id + "delete").click(function () { deleteBooking(booking._id) });
}
```

4.3. index.html

El menú principal se compone de dos partes:

- La primera parte se compone de tres entradas de datos (name, email y room_type) y un botón “POST Booking”, esto se utiliza para publicar una nueva reserva con los datos introducidos
- La segunda parte se compone de tres botones:
    - “GET Bookings” el cual mostrará todas las reservas.
    - “DELETE Bookings” con el cual borraremos por completo toda la base de datos.
    - “GET Booking” junto con una entrada para el id, este botón al ser pulsado leerá el id introducido y lo buscará en la base de datos para mostrarlo.

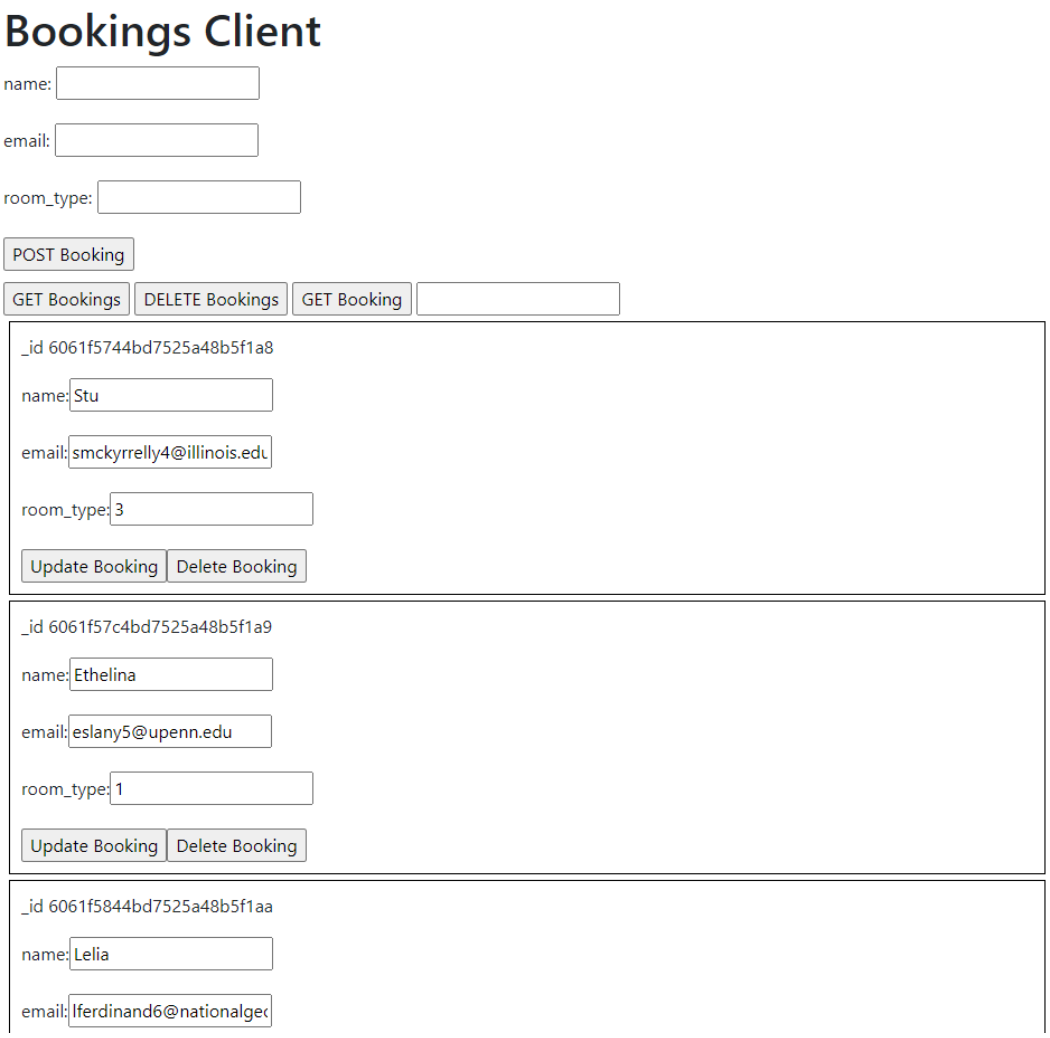
```
    <div class="bookings_menu">
        <h1>Bookings Client</h1>

        <div class="post">
            <div>
                <p>
                <label for="name_post">name: </label>
                <input type="text" id="name_post">
                </p>
                <p>
                    <label for="email_post">email: </label>
                    <input type="text" id="email_post">
                </p>
                <p>
                    <label for="room_type_post">room_type: </label>
                    <input type="text" id="room_type_post">
                </p>
            </div>
            <div>
                <input type="button" value="POST Booking" onclick="postBooking()" />
            </div>
        </div>
        <div>
            <input type="button" value="GET Bookings" onclick="getAllBookings()" />
            <input type="button" value="DELETE Bookings" onclick="deleteBookings()" />
            <input type="button" value="GET Booking" id="GetBookingButton" />

            <input type="text" name="_idIpt" id="_idInput">

            <span id="resBooking"></span>
        </div>
    </div>
</body>
<script>
    $("#GetBookingButton").click(function () { getBooking($("#_idInput").val()) });
</script>
```

Resultado de la operación **GET all** tras llamar a la función **jsonsToHtml**:



Como podemos ver, cada reserva irá acompañada de un botón de “Update” y de “Delete para actualizar o borrar la reserva según se necesite.