Sistemas Distribuidos e Internet

Servicios Web Rest

Sesión- 10.2

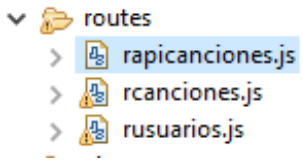
Curso 2017/ 2018

Implementación de una API Rest

En esta práctica vamos a crear una API de servicios web REST sobre la aplicación **tiendaCanciones**. Esta API nos permitirá gestionar las canciones, aunque utilizaremos una versión parcial de la entidad canción (sin fichero .mp3 y portada).

Gestión del recurso canción

Creamos un nuevo fichero **rapicanciones.js** en el vamos a implementar un API REST.



La URL **GET /api/cancion/** va a retornar una lista con todas las canciones**.**

Como se trata de un SW debemos retornar un código de respuesta estándar adecuado **res.status** https://es.wikipedia.org/wiki/Anexo:C%C3%B3digos_de_estado_HTTP y una respuesta en formato que pueda ser fácilmente procesado por una aplicación, por ejemplo en formato JSON (aunque también podrían servirnos otros como XML).

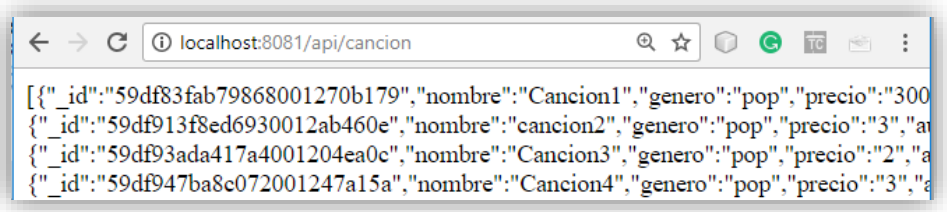
En este caso la gestión de las respuestas va a ser bastante directa ya que estamos utilizando una base de datos que ya tiene todos sus datos en JSON. Si no fuera así deberíamos aplicar obligatoriamente **JSON.stringfly** para transformarlos a objetos a JSON. Con **res.json** también podemos transformar objetos a formato JSON.

```
module.exports = function(app, gestorBD) {

    app.get("/api/cancion", function(req, res) {
        gestorBD.obtenerCanciones( {} , function(canciones) {
            if (canciones == null) {
                res.status(500);
                res.json({
                    error : "se ha producido un error"
                })
            } else {
                res.status(200);
                res.send( JSON.stringify(canciones) );
            }
        });
    });

}
```

Incluimos el nuevo controlador en **app.js**

```
//Rutas/controladores por lógica
require("./routes/rusuarios.js")(app, swig, gestorBD);
require("./routes/rcanciones.js")(app, swig, gestorBD);
require("./routes/rapicanciones.js")(app, gestorBD);
```

Probamos el servicio realizando una petición con el navegador http://localhost:8081/api/cancion



HATEOAS (Hypermedia as the Engine of Application State). Dentro de lo posible los clientes tienen que poder ir explorando los recursos de la aplicación. Cuando desde un recurso se haga referencia a otro se deben utilizar sus identificadores y a ser posible un enlace directo para explorar ese recurso.
Por ejemplo, la opción 2 sería mucho mejor que la 1, ya que es un enlace directo al usuario.

1. "autor" : prueba@prueba2.com
2. "autor" : /usuario/prueba2@prueba2.com

La URL **GET /api/cancion/:id** debería retornar la canción con el id correspondiente.

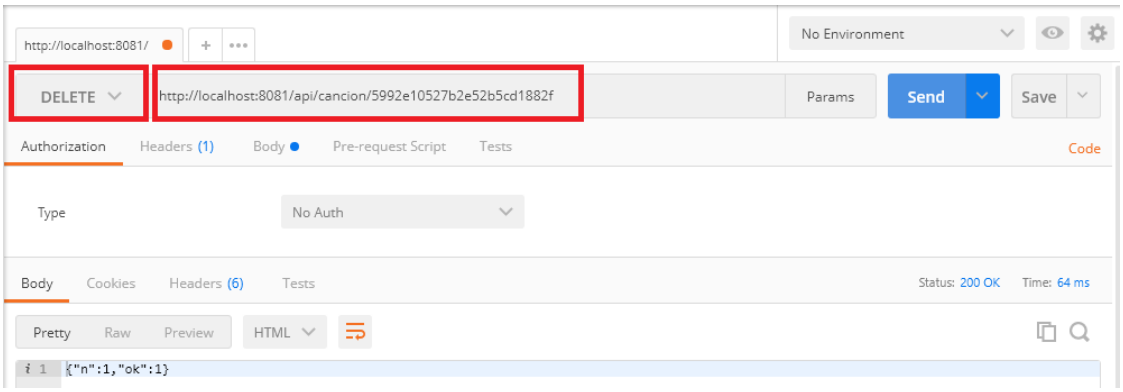
```
app.get("/api/cancion/:id", function(req, res) {
    var criterio = { "_id" : gestorBD.mongo.ObjectID(req.params.id)}

    gestorBD.obtenerCanciones(criterio,function(canciones){
        if ( canciones == null ){
            res.status(500);
            res.json({
                error : "se ha producido un error"
            })
        } else {
            res.status(200);
            res.send( JSON.stringify(canciones[0]) );
        }
    });
});
```

El servicio de eliminar canción utilizará el método **HTTP DELETE**, la petición será del estilo **DELETE /api/cancion/:id** . El resto de la petición será casi idéntica a la anterior.

```
app.delete("/api/cancion/:id", function(req, res) {
    var criterio = { "_id" : gestorBD.mongo.ObjectID(req.params.id)}

    gestorBD.eliminarCancion(criterio,function(canciones){
        if ( canciones == null ){
            res.status(500);
            res.json({
```

```
                error : "se ha producido un error"
            })
        } else {
            res.status(200);
            res.send( JSON.stringify(canciones) );
        }
    });
});
```

Para probar el funcionamiento del servicio DELETE podemos utilizar la aplicación **Postman** https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop (Una vez instalada podemos ver todas nuestras extensiones desde introduciendo la URL: chrome://apps)



Observamos que estamos devolviendo directamente la respuesta de la base de datos *{"n":1, "ok":1}* relativa al número de documentos afectados por el borrado, podríamos colocar otra respuesta más significativa.

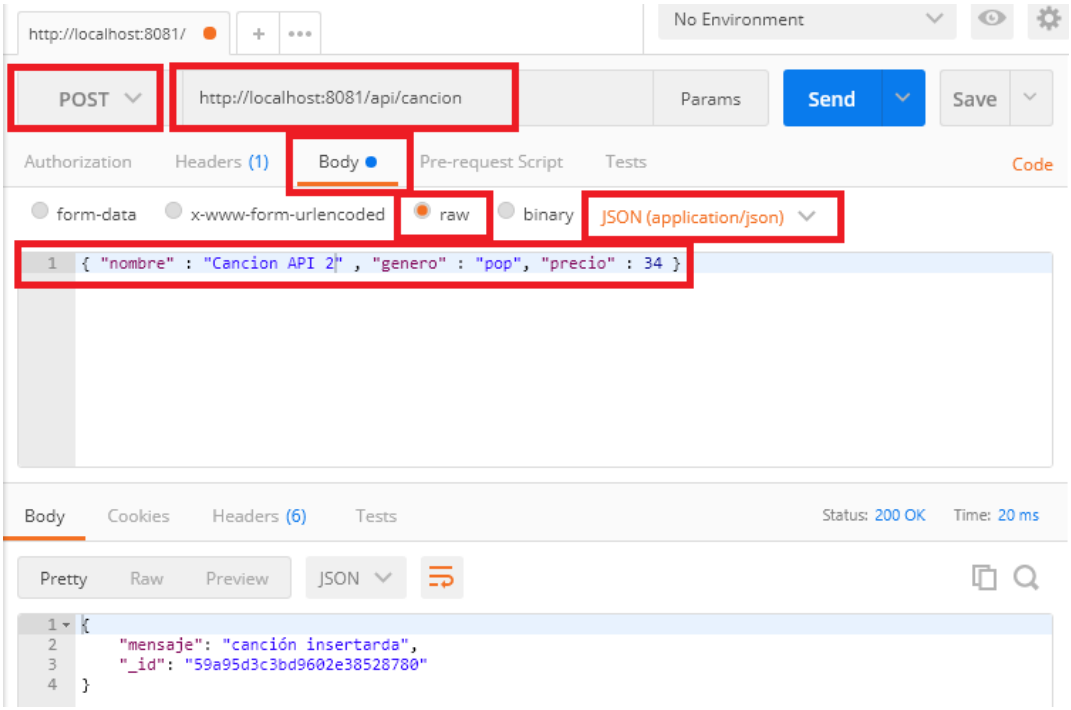
Para agregar un recurso canción utilizaremos la URL **POST /cancion**. La respuesta a la petición incluirá el código de http **201 – Created**, además de un mensaje en JSON.

```
app.post("/api/cancion", function(req, res) {
    var cancion = {
        nombre : req.body.nombre,
        genero : req.body.genero,
        precio : req.body.precio,
    }

    // ¿Validar nombre, genero, precio?

    gestorBD.insertarCancion(cancion, function(id){
        if (id == null) {
            res.status(500);
            res.json({
                error : "se ha producido un error"
            })
        } else {
            res.status(201);
            res.json({
                mensaje : "canción insertarda",
                 id : id
            })
        }
    });

});
```

*Deberíamos realizar validaciones en los datos (aunque nosotros no vamos a implementarlo):

1. La petición contiene todos los datos que se esperan
2. Los datos están en el formato correcto y dentro de los valores esperados, por ejemplo, en esta aplicación los géneros de las canciones (podrían estar limitados), el precio es un número positivo, el título no excede de N caracteres, etc.

Para asegurarnos de que funciona probamos el servicio utilizando **Postman**, debemos agregar un cuerpo **Body** a la petición en formato **raw** - **JSON** con el contenido: *{ "nombre" : "Cancion API 2" , "genero" : "pop", "precio" : 34 }*



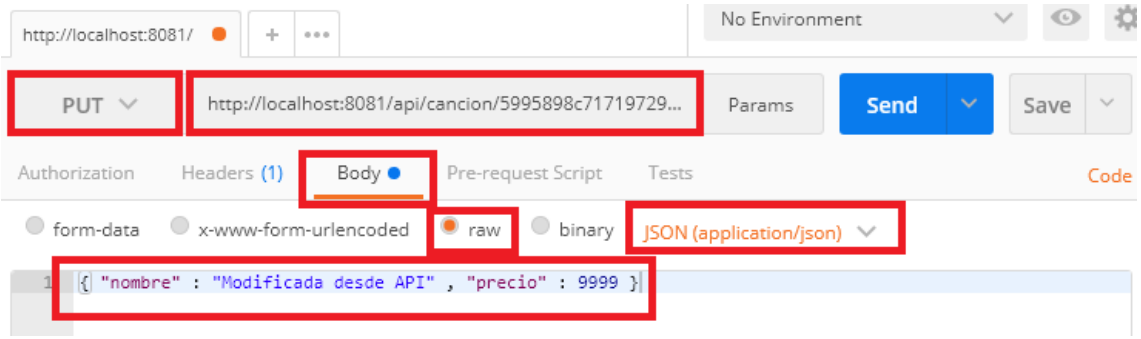
La función de modificar una canción será muy similar a la de crear canción, utilizaremos el método **PUT** (aunque también podríamos utilizar **UPDATE**).

En la URL debe aparecer el identificador de la canción que se va a modificar **PUT /api/cancion/:id** en el cuerpo de la petición se enviaran los datos que se deseen modificar (el cliente podría querer modificar solo una propiedad o varias).

```
app.put("/api/cancion/:id", function(req, res) {

    var criterio = { "_id" : gestorBD.mongo.ObjectID(req.params.id)  };

    var cancion = {}; // Solo los atributos a modificar
    if ( req.body.nombre != null)
        cancion.nombre = req.body.nombre;
    if ( req.body.genero != null)
        cancion.genero = req.body.genero;
    if ( req.body.precio != null)
        cancion.precio = req.body.precio;

    gestorBD.modificarCancion(criterio, cancion, function(result) {
        if (result == null) {
            res.status(500);
```

```
            res.json({
                error : "se ha producido un error"
            })
        } else {
            res.status(200);
            res.json({
                mensaje : "canción modificada",
                _id : req.params.id
            })

        }
    });
});
```

Al igual que en el caso de la creación de un nuevo recurso sería buena idea validar los datos.
Probamos la modificación desde el **Postman**, modificando parcialmente una de las canciones.



Identificación del cliente con Token y control de acceso

Las aplicaciones que ofrecen servicios web REST no acostumbran a hacer uso del objeto sesión, la identificación de los usuarios se lleva a cabo comúnmente utilizando un token de seguridad que identifica al cliente que realiza la petición. Este token se adjunta en todas las peticiones realizadas por el cliente, suele enviarse como parámetro GET, POST o en los HEADERS (siento este último el lugar más común).

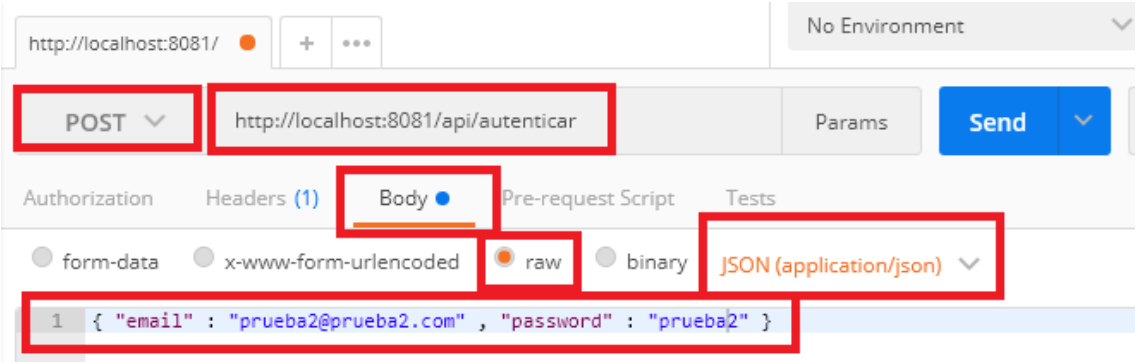
Aunque existen otros mecanismos de identificación basados en tokens la mayoría de APIs REST optan por uno de los siguientes:

1. **Token único,** cada cuenta de usuario es provista de un token único que le identifica, este token debe ser incluido en todas las peticiones realizadas a los servicios. De esta forma la aplicación identifica al cliente, además puede controlar si tiene permisos o no para ejecutar los servicios, el número de veces que llama a los servicios, etc.
2. **Token por login**, cuando el cliente envía sus credenciales a un servicio de identificación específico recibe un token, este token debe ser almacenado y enviado en todas las peticiones que el cliente realiza. Dependiendo de la implementación del servicio este token pueden funcionar de diferente manera, por ejemplo, caducando después de un tiempo sin recibir peticiones (algo similar a la sesión) o incluso caducando en cada petición y reenviando siempre el servidor uno nuevo como parte de todas las respuestas.

Vamos a implementar la opción (2) **Token por login**. Creamos un servicio para identificar al usuario asociado a la URL **POST /autentificar**, muy similar al login que implementamos en la

aplicación web. A partir del **email** y el **password encriptado** del usuario se realiza una búsqueda en la base de datos, si hay coincidencia retornamos un JSON con el parámetro autenticado a true, si no la hay retornamos un false.

```
app.post("/api/autenticar/", function(req, res) {
    var seguro = app.get("crypto").createHmac('sha256', app.get('clave'))
    .update(req.body.password).digest('hex');

    var criterio = {
        email : req.body.email,
        password : seguro
    }

    gestorBD.obtenerUsuarios(criterio, function(usuarios) {
        if (usuarios == null || usuarios.length == 0) {
            res.status(401); // Unauthorized
            res.json({
                autenticado : false
            })
        } else {
            res.status(200);
            res.json({
                autenticado : true
            })
        }

    });
});
```



Debemos modificar la aplicación para que devuelva un **token de seguridad** cuando la autentificación sea correcta, hay varias estrategias que podemos seguir para crear el token:

1. Crear un token totalmente aleatorio y almacenarlo junto a la información del usuario en la base de datos.
2. Crear un token con el identificador del usuario encriptado (y que nosotros podamos desencriptar para saber de qué usuario se trata). Además del identificador del usuario el token puede contener otra información como fecha de creación del token (para controlar su expiración).
3. Otras muchas variaciones para incrementar el nivel de seguridad, como por ejemplo incluir la IP del cliente

Optaremos por la (2) encriptamos el email del **usuario + milisegundo actual Timestamp Date.now() ,** incluir el tiempo actual es muy útil para implementar caducidades. Aunque

podríamos utilizar el módulo crypto para realizar estas encriptaciones vamos a optar por usar el módulo **jsonwebtoken** https://www.npmjs.com/package/jsonwebtoken , ya que es bastante popular para realizar este tipo de sistemas.

Descargamos el módulo accediendo desde la consola de comandos al directorio raíz del proyecto y ejecutando: **npm install jsonwebtoken --install**

```
C:\Users\jordansoy\work\TiendaMusica>npm install jsonwebtoken --install
TiendaMusica@0.1.0 C:\Users\jordansoy\work\TiendaMusica
`-- jsonwebtoken@8.1.0
  +-- jws@3.1.4
  | +-- base64url@2.0.0
  | `-- jwa@1.1.5
  |   +-- buffer-equal-constant-time@1.0.1
  |   `-- ecdsa-sig-formatter@1.0.9
  +-- lodash.includes@4.3.0
  +-- lodash.isboolean@3.0.3
```

Dentro del fichero principal de la aplicación **app.js** declaramos el require del nuevo módulo, el cual vamos a almacenar en las variables de la aplicación, bajo la clave "**jwt**" (De esta forma podemos acceder a la variable **jwt** desde cualquier parte de la aplicación)

```
var express = require('express');
var app = express();

var jwt = require('jsonwebtoken');
app.set('jwt',jwt);
```

Modificamos la respuesta de **POST /api/autenticar** implementada en **raplicaciones.js** . Cuando los credenciales sean correctos generamos un token, almacenando en él: el **email del usuario** y el **tiempo** (como Date.now() devuelve milisegundos actual, vamos a convertirlo a segundos dividiéndolo entre 1000). Retornamos el nuevo token como respuesta.

```
app.post("/api/autenticar/", function(req, res) {
    var seguro = app.get("crypto").createHmac('sha256', app.get('clave'))
    .update(req.body.password).digest('hex');

    var criterio = {
        email : req.body.email,
        password : seguro
    }

    gestorBD.obtenerUsuarios(criterio, function(usuarios) {
        if (usuarios == null || usuarios.length == 0) {
            res.status(401);
            res.json({
                autenticado : false
            })
        } else {
            var token = app.get('jwt').sign(
                    {usuario: criterio.email , tiempo: Date.now()/1000},
                    "secreto");
            res.status(200);
            res.json({
                autenticado: true,
                token : token
            });
        }

    });
```

```
});
```

Vamos a requerir que la petición incluya el token para cualquier llamada a **/api/cancion/**

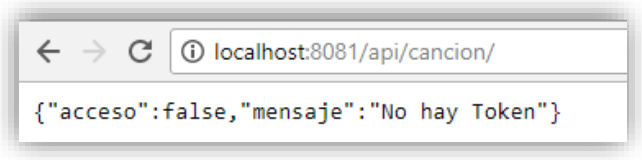
Implementamos en nuevo Router **routerUsuarioToken** en el fichero principal **app.js**

- Obtenemos el parámetro **token**, admitimos que se envié, como parámetro POST, GET o HEADER (cualquiera de las ubicaciones nos sirve)
- Verificamos el token desencriptándolo
  - Si no conseguimos desencriptarlo o si han pasado más de 240 segundos (Date.now()/100 – tokenInfo.tiempo > 240) desde que se creó el token retornamos un mensaje de error: "Token invalido o caducado"
  - Si lo desencriptamos correctamente dejamos correr la petición **next().** Puede ser buena idea guardar el identificador del usuario la respuesta, **res.usuario** , de esta forma no habrá que volver a des encriptar el token más adelante si queremos saber del que usuario se trata.
- Si nisiquiera hay token, enviamos un mensaje de error "No hay token".

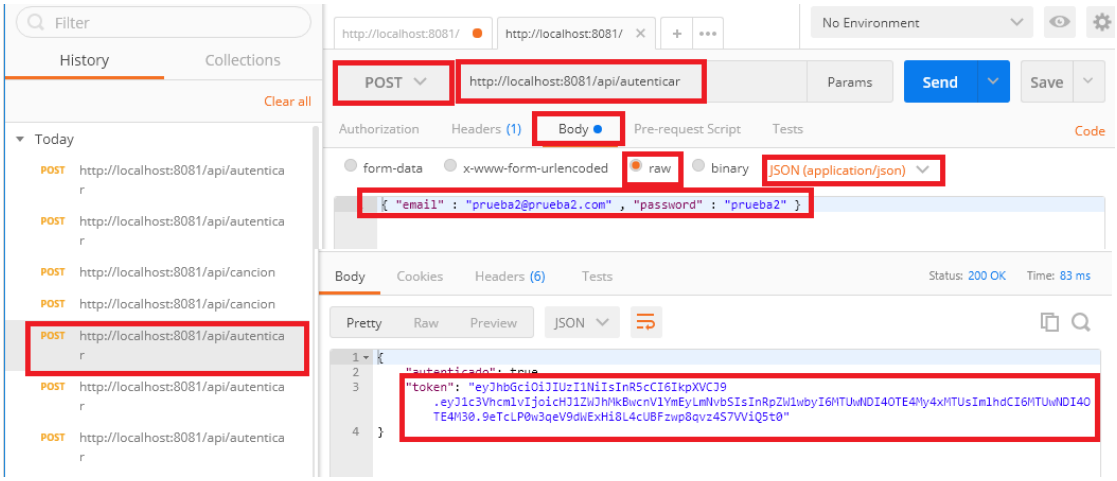
```
var gestorBD = require("./modules/gestorBD.js");
gestorBD.init(app,mongo);

// routerUsuarioToken
var routerUsuarioToken = express.Router();
routerUsuarioToken.use(function(req, res, next) {
          // obtener el token, puede ser un parámetro GET , POST o  HEADER

          var token = req.body.token || req.query.token || req.headers['token'];
          if (token != null) {
              // verificar el token
                  jwt.verify(token, 'secreto', function(err, infoToken) {
                      if (err || (Date.now()/1000 - infoToken.tiempo) > 240 ){
                          res.status(403); // Forbidden
                          res.json({
                                  acceso : false,
                                  error: 'Token invalido o caducado'
                          });
                          // También podríamos comprobar que intoToken.usuario existe
                          return;

                      } else {
                        // dejamos correr la petición
                        res.usuario = infoToken.usuario;
                        next();
                      }
                  });

          } else {
                  res.status(403); // Forbidden
                  res.json({
                      acceso : false,
                      mensaje: 'No hay Token'
                  });
          }
});

// Aplicar routerUsuarioToken
app.use('/api/cancion', routerUsuarioToken);

// routerUsuarioSession
var routerUsuarioSession = express.Router();
routerUsuarioSession.use(function(req, res, next) {
```

```
        console.log("routerUsuarioSession");
         if ( req.session.usuario ) {
           // dejamos correr la petición
            next();
         } else {
            req.session.destino = req.originalUrl;
            console.log("va a : "+req.session.destino)
            res.redirect("/identificarse");
         }
});
```

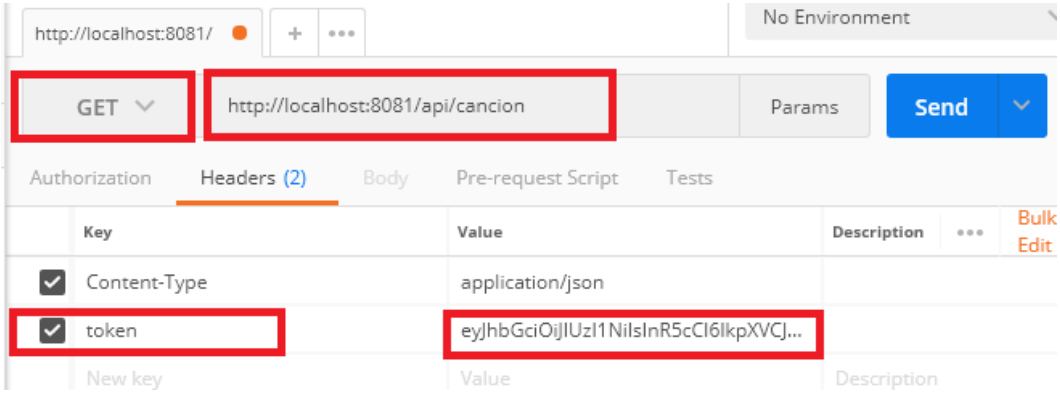
Guardamos los cambios y ejecutamos la aplicación, probamos a acceder al servicio http://localhost:8081/api/cancion desde el navegador.



Para probar en detalle todo lo implementado necesitaremos utilizar el **PostMan** , primero hacemos una petición a **POST** http://localhost:8081/api/autenticar (Incluyendo en el body el **email** y **password**), si la petición es correcta nos retornará el **token**.



Copiamos el **token** y creamos una nueva petición dirigida a GET http://localhost:8081/api/cancion , incluimos una nueva cabecera **Headers**, con clave **token** y valor el token obtenido.



Una vez incluimos el token en nuestras peticiones podemos utilizar todos los servicios de **/api/cancion** (lo verificamos)

****Nota,** aunque nosotros no vamos a incluirlo, habría que asegurarse de que el usuario no solo está identificado, sino que es el dueño de la canción cuando se trata de eliminar o modificar canciones. Esta comprobación se puede añadir en las propias funciones que responden a eliminar y modificar (*recomendado) o creando un nuevo router.

Posibles mejoras

La API de servicios web implementada tiene una funcionalidad no completa, por ejemplo:

- No se ha implementado un servicio para obtener únicamente las canciones publicadas por el usuario identificado en el token.
- No se han agregado las comprobaciones de que el usuario que intenta **eliminar** o **modificar** una canción es su autor (el usuario se identifica a través del token).