



# **Sistemas Distribuidos e Internet**

## **Desarrollo de aplicaciones Web con NodeJS**

### **Sesión- 8**

### **Curso 2017/ 2018**



## Acceso a datos de Mongo desde Node.js

Para conectarnos a la base de datos desde la aplicación utilizaremos el módulo **mongodb** <https://docs.mongodb.com/ecosystem/drivers/node-js/>. Se trata de un módulo externo que nos permite conectarnos fácilmente a mongo, pero no es la única alternativa disponible en Node.

Abrimos la consola de comandos y nos situamos en el directorio raíz del proyecto.

Ejecutamos **npm install mongodb@2.2.33 --save**

Es importante instalar la versión 2.\* ya que las versiones 2.\* y 3.\* se mantienen en paralelo y utilizan un conjunto de funciones totalmente diferentes.

```
\WorkingWithNodeJS\TiendaMusica>npm install mongodb@2.2.33 --save
TiendaMusica@0.1.0 C:\Users\virtual_user\Source\Repositories\workspac
e-sts-3.8.4.RELEASE\WorkingWithNodeJS\TiendaMusica
+-- mongodb@2.2.33
+-- es6-promise@3.2.1
+-- mongodb-core@2.1.17
| +-- bson@1.0.4
| +-- require_optional@1.0.1
| +-- resolve-from@2.0.0
| +-- semver@5.4.1
+-- readable-stream@2.2.7
```

Para poder acceder al módulo mongo incluimos la sentencia **require 'mongodb'**. Agregamos este objeto en el fichero principal **app.js**. Obtenemos la referencia a mongo y la enviaremos como parámetro al controlador **rcanciones** (igual que hicimos con swig).

También creamos una variable con clave **set.db('db', <valor>)** en la que introducimos la **cadena de conexión**

- **Recomendado:** usar la cadena de conexión obtenida en nuestro mLab (mongo desplegado en la nube [cada uno la suya](#)) Revisar la parte final del guión anterior  
`mongodb://admin:sdi@ds117935.mlab.com:17935/tiendamusica`
- **Alternativa:** usar la cadena de conexión del servidor local de mongo que tenemos en nuestra máquina  
`mongodb://localhost:27017/uomusic`

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

var mongo = require('mongodb');
var swig = require('swig');
var bodyParser = require('body-parser');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static('public'));

// Variables
```



```
app.set('port', 8081);
app.set('db', 'mongodb://localhost:27017/uomusic');

//Rutas/controladores por logica
require('./routes/rusuarios.js')(app, swig);
require('./routes/rcanciones.js')(app, swig, mongo);
```

### SUSTITUIR LA CADENA DE CONEXIÓN POR LA PROPIA

Recibimos el parámetro **mongo** en el controlador **rcanciones.js**

```
module.exports = function(app, swig, mongo) {
```

La función que procesa la petición enviada por el formulario ya está implementada (es **POST /canción**), ahora vamos a crear un **objeto** canción con parámetros recibidos.

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

Ya tenemos definido el objeto que queremos incluir, ahora hacer falta conectarse a la base de datos e insertarlo.

La función **connect()** del objeto **MongoClient** nos sirve para establecer conexiones, esta función recibe como parámetros:

1. La cadena de conexión, la cual tenemos almacenada en la variable de aplicación 'db'.
2. La función que se ejecutará al completar la conexión. Esta función tiene dos parámetros:
  - **err** : es un objeto donde se almacenan los errores, es nulo/vacío si no ha habido ningún problema.
  - **db** : es una referencia a la base de datos, sobre este objeto se pueden ejecutar las acciones (insertar, consultar, etc.).

En la función que enviamos como parámetro utilizaremos la conexión a la base de datos para insertar el objeto **cancion**. Recuperamos la colección canciones con **db.collection(canciones)**, sobre esta colección ejecutamos una inserción, con la función **insert**.

La función **insert** recibe como parámetros:

1. El propio objeto a insertar, **cancion**
2. Una función que se ejecuta al finalizar la operación insert, desde esta función debemos ejecutar la función **db.close()** para cerrar la conexión a la base de datos. Esta función recibe otros dos parámetros
  - **err** : es un objeto donde se almacenan los errores, es nulo/vacío si no ha habido ningún problema.



- **result** : es un array con los objetos que ha sido insertado en la base de datos. Insert agrega a los documentos una propiedad automática “\_id” . Si insertamos un solo documento y queremos consultar su \_id debemos acceder a **result.ops[0].\_id** .

En la implementación de la conexión e inserción del dato es importante dar diferentes respuestas (res.send()) cuando se produce un error.

```
app.post("/cancion", function(req, res) {  
  var cancion = {  
    nombre : req.body.nombre,  
    genero : req.body.genero,  
    precio : req.body.precio  
  }  
  // Conectarse  
  mongo.MongoClient.connect(app.get('db'), function(err, db) {  
    if (err) {  
      res.send("Error de conexión: " + err);  
    } else {  
      var collection = db.collection('canciones');  
      collection.insert(cancion, function(err, result) {  
        if (err) {  
          res.send("Error al insertar " + err);  
        } else {  
          res.send("Agregada id: " + result.ops[0]._id);  
        }  
        db.close();  
      });  
    }  
  });  
});
```

\*Importante, debemos tener en cuenta que el código de **MongoClient.connect** (como el de muchas funciones en JavaScript y Node.js) se ejecuta de forma asíncrona mediante un sistema de callbacks.



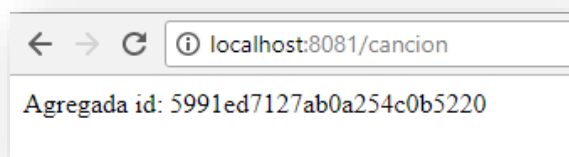
```
app.post("/cancion", function(req, res) {
  var cancion = {
    nombre : req.body.nombre,
    genero : req.body.genero,
    precio : req.body.precio
  }

  // Conectarse
  mongo.connect(app.get('db'), function(err, db) {
    if (err) {
      res.send("Error de conexión: " + err);
    } else {
      var collection = db.collection('canciones');
      collection.insert(cancion, function(err, result) {
        if (err) {
          res.send("Error al insertar " + err);
        } else {
          res.send("Agregada id: " + result.ops[0]._id);
        }
      });
      db.close();
    }
  });
});
```

Agregamos una canción y comprobamos que funciona correctamente, debemos asegurarnos de que la base de datos está arrancada. Accedemos a <http://localhost:8081/canciones/agregar> y completamos el formulario.

**\*\*Nota:** asegurarse de que la función `app.get("canciones/:id")` no está declarada antes que `app.get("canciones/agregar")`; porque de lo contrario entrará por ella, suponiendo que "agregar" es la ID.

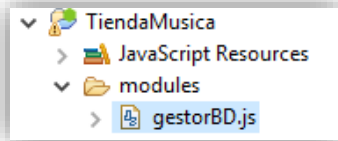
Una vez agregada la respuesta será del tipo:



## Arquitectura para el acceso a datos

Aunque la implementación anterior funciona correctamente nos podría originar problemas de mantenibilidad y reutilización en aplicaciones grandes. Una buena decisión sería sacar la gestión de la base de datos a un **módulo-objeto** independiente.

Creamos una nueva carpeta **modules** en el directorio raíz, dentro de ella un fichero **gestorBD.js**



A diferencia de los modules anteriores **rcanciones** y **rusuarios**, este módulo va a ser un **OBJETO**, con variables y funciones a las que nosotros decidimos cuando llamar (los otros dos módulos que habíamos creado eran una función que se ejecutaba de forma automática al hacer el require).

**NOTA:** la sintaxis dentro de un objeto va a ser significativamente distinta en JS.

Diferencias:

- Las variables globales se declaran usando *nombre : valor*
- Todos los elementos están separados por comas.
- Las funciones se declaran *nombre : function (parámetros)*
- **Hay que poner this. Para acceder a las variables locales.**

Hemos creado dos variables: **app** y **mongo**.

Una función **init(app,mongo)** para inicializar las dos variables globales. Una función insertarCancion que al igual que antes funciona de forma de forma **Asíncrona, por lo que no puede tener return**, para retornar el dato recibe una **función de callback**, es decir cuando ya ha conseguido un valor se lo envía a esa función de callback.

- En caso de éxito la función de callback recibe la ID de la canción insertada
- En caso de error la función de callback recibe un null

```
module.exports = {
  mongo : null,
  app : null,
  init : function(app, mongo) {
    this.mongo = mongo;
    this.app = app;
  },
  insertarCancion : function(cancion, functionCallback) {
    this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {
      if (err) {
        functionCallback(null);
      } else {
        var collection = db.collection('canciones');
        collection.insert(cancion, function(err, result) {
          if (err) {
            functionCallback(null);
          } else {
            functionCallback(result.ops[0]._id);
          }
        });
        db.close();
      }
    });
  }
};
```



Para utilizar este objeto tenemos que agregarlo en **app.js** con un **require**, y llamar a su función **init(app,mongo)** . Hay que asegurarse de agregar el módulo después del módulo **mongo** puesto que lo recibe como parámetro.

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

var mongo = require('mongodb');
var swig = require('swig');
var bodyParser = require('body-parser');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

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

Después vamos a enviar el nuevo objeto **gestorBD** a nuestros dos controladores.

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

Indicamos que este parámetro se va a recibir en las funciones **rusuarios**

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

Y de igual manera en **rcanciones**.

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

Modificamos el **POST /canción** para utilizar el **gestorBD**.

```
app.post("/cancion", function(req, res) {
  var cancion = {
    nombre : req.body.nombre,
    genero : req.body.genero,
    precio : req.body.precio
  }
  // Conectarse
  gestorBD.insertarCancion(cancion, function(id){
    if (id == null) {
      res.send("Error al insertar ");
    } else {
      res.send("Agregada id: " + id);
    }
  });
});
```



## Subida de ficheros

Vamos a implementar la lógica asociada a la subida de la imagen y el fichero de audio.

Para poder subir ficheros debemos descargar el módulo externo **express-fileupload**.

Accedemos por consola de comandos al directorio raíz del proyecto y ejecutamos el comando **npm install express-fileupload --save**

```
C:\Users\jordansoy>cd C:\Users\jordansoy\work\TiendaMusica
C:\Users\jordansoy\work\TiendaMusica>npm install express-fileupload --save
TiendaMusica@0.1.0 C:\Users\jordansoy\work\TiendaMusica
-- express-fileupload@0.2.0
+- busboy@0.2.14
| +- dicer@0.2.5
| | +- readable-stream@1.1.14
| | | +- isarray@0.0.1
| | | +- string_decoder@0.10.31
| | | -- streamsearch@0.1.2
| | -- readable-stream@1.1.14
| +- isarray@0.0.1
| -- string_decoder@0.10.31
```

Añadimos el módulo en el fichero principal **app.js**, en este caso no es necesario enviarlo como parámetro a los controladores ya que este objeto no se usa directamente, sino que se integra en la aplicación (app.use).

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

var fileUpload = require('express-fileupload');
app.use(fileUpload());
var mongo = require('mongodb');
```

Modificamos la petición **POST /canciones**, una vez este guardada la canción en la base de datos no vamos a enviar la respuesta con **res.send()**. En lugar de responder realizaremos la subida de los ficheros, cuando la subida esté finalizada se retornará la respuesta.

Comprobamos si en la request ha llegado un fichero con el nombre portada (**req.files.portada.name**) , si es así guardamos el fichero en una variable y lo salvamos con la función **mv(path, función de callback)**. Una vez se ejecuta la función de callback significa que el fichero ya ha sido guardado, dependiendo del contenido de la variable **err** retornamos una respuesta u otra. En el fichero **rcanciones** modificamos el método **gestorBD.insertarCancion()**.

```
gestorBD.insertarCancion(cancion, function(id) {
  if (id == null) {
    res.send("Error al insertar ");
  } else {
    res.send("Agregada id: " + id);
    if (req.files.portada != null) {
      var imagen = req.files.portada;
      imagen.mv('public/portadas/' + id + '.png', function(err) {
        if (err) {
          res.send("Error al subir la portada");
        } else {
          res.send("Agregada id: " + id);
        }
      });
    }
  }
});
```

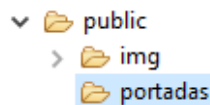




```
}  
  
});
```

Asegurarse de borrar la línea **res.send()** porque no se puede seguir procesando la petición una vez se ha enviado la respuesta.

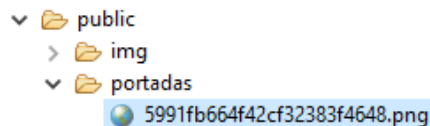
Como hemos indicado que el fichero se guarda en la carpeta **public/portadas** debemos crearla (sino el proceso fallara).



Antes de probarlo debemos asegurarnos de que el formulario contiene la propiedad **enctype="multipart/form-data"**. Abrimos la vista correspondiente **/views/bagregar.html** y añadimos la propiedad en la declaración.

```
<h2>Agregar canción</h2>  
<form class="form-horizontal" method="post" action="/cancion" enctype="multipart/form-data">
```

Guardamos los cambios y ejecutamos la aplicación, probamos a agregar una nueva canción con una imagen como portada. Si actualizamos el eclipse (F5) veremos cómo aparecen los ficheros que acabamos de subir.



Para subir el fichero de audio el proceso va a ser muy similar, una vez se ha subido con éxito la portada comprobamos si la petición contiene **req.files.audio.name**.

```
if (req.files.portada != null) {  
    var imagen = req.files.portada;  
    imagen.mv('public/portadas/'+id+'.png', function(err) {  
        if (err) {  
            res.send("Error al subir la portada");  
        } else {  
            res.send("Agregada id: "+result.ops[0]._id);  
            if (req.files.audio != null) {  
                var audio = req.files.audio;  
                audio.mv('public/audios/'+id+'.mp3', function(err) {  
                    if (err) {  
                        res.send("Error al subir el audio");  
                    } else {  
                        res.send("Agregada id: "+ id);  
                    }  
                });  
            }  
        }  
    });  
}
```



```
}  
});  
}
```

Asegurarse de borrar la línea **res.send()** porque no se puede seguir procesando la petición una vez se ha enviado la respuesta. También hay que asegurarse de que la función siempre ofrece una respuesta.

Creamos el directorio **/public/audios** para que los ficheros puedan ser subidos.

```
✓ public  
  audios  
  > img  
  > portadas
```

Deberíamos modificar el formulario de la vista **bagregar.html** e incluir la palabra **required** en los dos ficheros, también es buena idea acotar el tipo de formatos aceptados.

```
<div class="form-group">  
  <label class="control-label col-sm-2" for="portada">Imagen portada:</label>  
  <div class="col-sm-10">  
    <input type="file" class="custom-file-input" name="portada"  
      accept=".png" required/>  
  </div>  
</div>  
<div class="form-group">  
  <label class="control-label col-sm-2" for="audio">Fichero audio:</label>  
  <div class="col-sm-10">  
    <input type="file" class="custom-file-input" name="audio"  
      accept=".mp3" required/>  
  </div>  
</div>
```

Guardamos los cambios y probamos a ejecutar la aplicación. En la carpeta **canciones.zip** disponible en el campus virtual hay varios recursos que podemos utilizar para crear canciones.

Probamos a subir una canción completa.

## Recuperar y listar las canciones

Vamos a responder a la petición **GET /tienda** mostrando el catálogo de todas las canciones almacenadas en la base de datos. Para ello abrimos el objeto **gestorBD.js** y agregamos una nueva función **obtenerCanciones()**.

La función **obtenerCanciones** va a retornar todos los documentos almacenados en la colección "canciones" sin aplicar ninguna condición **.find()** (sin no establecemos un criterio nos retorna todas las canciones). El resultado de la consulta lo transformamos a un array.

```
module.exports = {  
  mongo : null,  
  app : null,  
  init : function(app, mongo) {
```

```

this.mongo = mongo;
this.app = app;
},
obtenerCanciones : function(functionCallback){
    this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {
        if (err) {
            functionCallback(null);
        } else {
            var collection = db.collection('canciones');
            collection.find().toArray(function(err, canciones) {
                if (err) {
                    functionCallback(null);
                } else {
                    functionCallback(canciones);
                }
            });
            db.close();
        }
    });
});
},
};

```

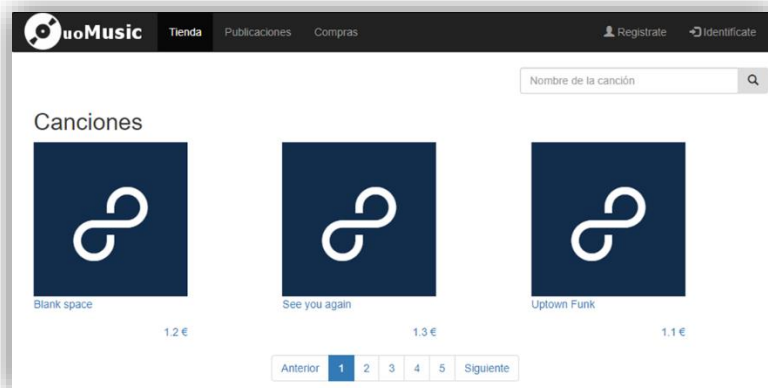
Agregamos la función **GET /tienda** en el fichero **rcanciones.js**, realizamos una llamada a **obtenerCanciones()**, una vez recibida la lista se la enviamos como atributo a la plantilla **btienda.html** bajo la clave **canciones**.

```

app.get("/tienda", function(req, res) {
    gestorBD.obtenerCanciones( function(canciones) {
        if (canciones == null) {
            res.send("Error al listar ");
        } else {
            var respuesta = swig.renderFile('views/btienda.html',
            {
                canciones : canciones
            });
            res.send(respuesta);
        }
    });
});

```

Sí guardamos los cambios y ejecutamos la aplicación podremos ver en <http://localhost:8081/tienda> el catálogo de canciones, previamente debimos guardarlas en la base de datos desde la propia aplicación web.





La vista aun no muestra las portadas de las canciones, debemos acceder a **views/btienda.html** y modificar el código HTML correspondiente a la imagen de la portada.

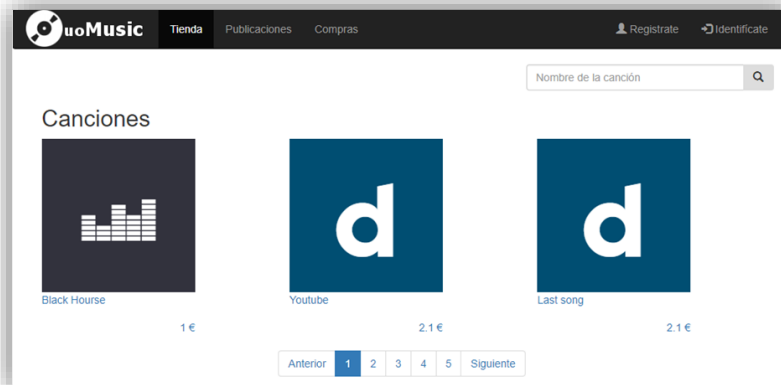
La imagen es un fichero png que se debe encontrar en la carpeta **/portadas/{id de la canción}.png** . Sin embargo las id de los documentos mongo son OBJETOS no cadenas de texto, por lo tanto no podemos utilizar directamente **\_id** debemos obtener el String con la función **toString()**.

Ejemplo de documentos Mongo, **\_id** es de tipo **ObjectId**, no es una cadena.

```
> db.canciones.find({});
{ "_id" : ObjectId("5992e0bc27b2e52b5cd1882e"), "nombre" : "Black Hourse", "genero" : "pop", "precio" : "1" }
{ "_id" : ObjectId("5992e10527b2e52b5cd1882f"), "nombre" : "Youtube", "genero" : "folk", "precio" : "2.1" }
{ "_id" : ObjectId("5992e12727b2e52b5cd18830"), "nombre" : "Last song", "genero" : "folk", "precio" : "2.1" }
```

```
{% for cancion in canciones %}
<div class="col-xs-6 col-sm-6 col-md-4 col-lg-3">
  <div style="width: 200px">
    <a href="/cancion/id"> 
    <!-- http://www.socicon.com/generator.php -->
    <div class="wrap">{{ cancion.nombre }}</div>
    <div class="small">Autor</div>
    <div class="text-right">{{ cancion.precio }} €</div>
  </a>
</div>
</div>
{% endfor %}
```

Guardamos los cambios y ejecutamos la aplicación <http://localhost:8081/tienda>



## Listar canciones por criterio

Vamos a introducir un nuevo parámetro en la función **obtenerCanciones()** del **gestorBD.jsp**. Este parámetro será la condición de búsqueda (criterio que deben de cumplir las canciones para ser retornadas, similar a la condición where). El criterio se envía como parámetro en la función **find(<criterio>)** . Si no quisiéramos ningun criterio el parámetro a enviar debria ser un objeto vacío **{}** , si quisiéramos la canción con nombre "Black House" el criterio sería: **{ "nombre": "Black House" }**



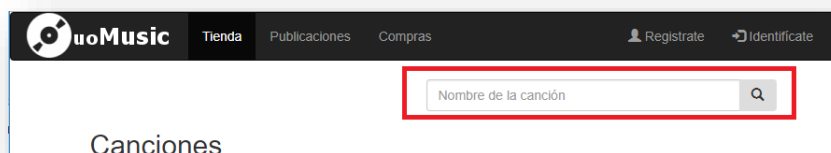
```
obtenerCanciones : function(criterio,functionCallback){
    this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {
        if (err) {
            functionCallback(null);
        } else {
            var collection = db.collection('canciones');
            collection.find(criterio).toArray(function(err, canciones) {
                if (err) {
                    functionCallback(null);
                } else {
                    functionCallback(canciones);
                }
            });
            db.close();
        }
    });
},
```

Para listar todas las canciones de la tienda en **GET /tienda** el criterio que aplicamos al llamar a la función **obtenerCanciones()** debe ser vacío {} .

```
app.get("/tienda", function(req, res) {
    var criterio = {};

    gestorBD.obtenerCanciones(criterio, function(canciones) {
        if (canciones == null) {
            res.send("Error al listar ");
        } else {
            var respuesta = swig.renderFile('views/btienda.html',
            {
                canciones : canciones
            });
            res.send(respuesta);
        }
    });
});
```

La búsqueda de la aplicación está pensada para buscar canciones por nombre. Este formulario envía una petición **GET /tienda?busqueda=<nombre>** , debemos procesar el parámetro búsqueda y mostrar en la vista únicamente las canciones que coincidan con ese criterio.



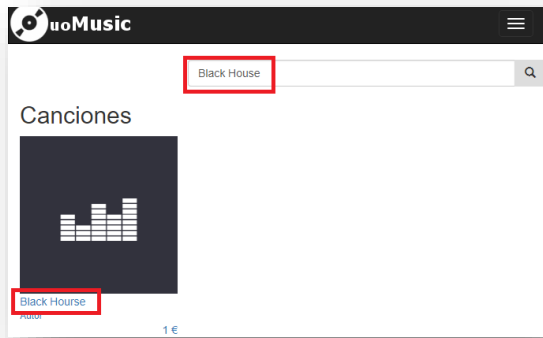
En el controlador **GET /tienda** comprobamos si nos ha llegado un parámetro **req.query.busqueda** , si no nos ha llegado el criterio de búsqueda será el objeto vacío {} (ningún criterio) , si nos ha llegado el parámetro el criterio será { **"nombre" : req.query.busqueda** }

```
app.get("/tienda", function(req, res) {
    var criterio = {};
```



```
if( req.query.búsqueda != null ){  
    criterio = { "nombre" : req.query.búsqueda };  
}
```

En este caso y con la condición que hemos aplicado en el find el nombre de la canción buscada debe coincidir exactamente con una de las canciones que tenemos almacenadas.



Podríamos aplicar comodines u otros criterios en la búsqueda:

- <https://docs.mongodb.com/manual/reference/method/db.collection.find/>
- <https://docs.mongodb.com/manual/reference/operator/query-logical/>

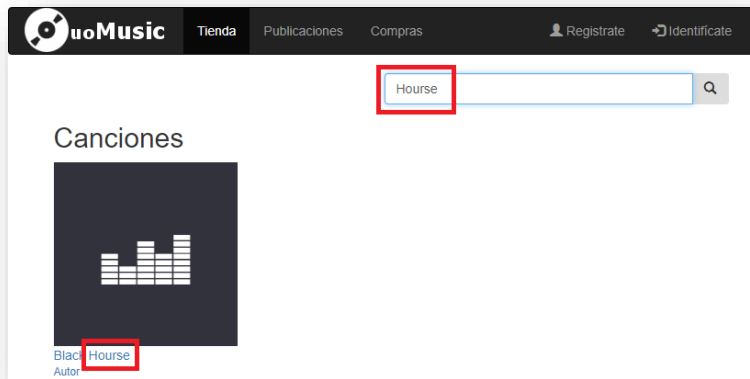
También expresiones regulares/patrones

- <https://docs.mongodb.com/manual/reference/operator/query/regex/>

Vamos a ampliar el criterio haciendo que los resultados tengan la cadena introducida en su nombre pero que puedan tener más caracteres antes y/o después.

```
app.get("/tienda", function(req, res) {  
    var criterio = {};  
    if( req.query.búsqueda != null ){  
        criterio = { "nombre" : { $regex : ".*"+req.query.búsqueda+".*" } };  
    }  
}
```

Guardamos los cambios, ejecutamos la aplicación y comprobamos el nuevo funcionamiento.



## Vista de detalles de la canción

Al pulsar sobre la miniatura de una canción en la tienda queremos que abra la vista en detalle.

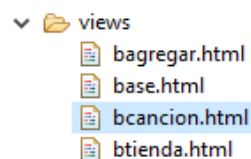
Utilizaremos el `_id` del objeto canción como identificador, de tal forma que la petición para ver los detalles de una canción será: **GET /cancion/{id}**. Añadimos a los detalles de la canción en la vista **/views/btienda.html**

```
{% for cancion in canciones %}
<div class="col-xs-6 col-sm-6 col-md-4 col-lg-3">
  <div style="width: 200px">
    <a href="/cancion/{{ cancion._id.toString() }}">
      
      <!-- http://www.socicon.com/generator.php -->
      <div>{{ cancion.nombre }}</div>
      <div class="small">{{ cancion.autor }}</div>
      <div class="text-right">{{ cancion.precio }} €</div>
    </a>
  </div>
</div>
{% endfor %}
```

La lógica de la función **GET /cancion/{id}** va a consistir en:

1. Obtener la canción que se corresponde con la **{id}**.
2. Enviar la canción a una vista para que la muestre.

Movemos el fichero **public/bcancion.html** que habíamos descargado del campus virtual a la carpeta **/views**.





Aunque ya tenemos la estructura HTML nos faltan todas las sentencias twig que muestran los valores de los atributos de la canción. No pasa nada por utilizar parámetros de la canción aun no creados, como **canción.autor**, simplemente no se mostrarán pero no producirán errores.

```
{% extends "base.html" %}

{% block titulo %} Detalles {{ cancion.nombre }} {% endblock %}

{% block contenido_principal %}
<div class="row">
  <div class="media col-xs-10">
    <div class="media-left media-middle">
      
    </div>
    <div class="media-body">
      <h2>{{ cancion.nombre }}</h2>
      <p>{{ cancion.autor }}</p>
      <p>{{ cancion.genero }}</p>
      <button type="button" class="btn btn-primary pull-right">{{ cancion.precio }} €</button>
      <!-- Cambiar el precio por "reproducir" si ya está comprada -->
    </div>
  </div>
</div>
</div>
```

Implementamos la respuesta **GET /cancion/{id}**, ejecutando la función que obtiene la canción de la base de datos y enviándole esa canción a la plantilla **bcancion.html**.

Para obtener la canción vamos a utilizar la función **obtenerCanciones()** aplicándole como criterio la **\_id** de la canción. Hay que tener en cuenta que **obtenerCanciones()** retorna un array de canciones, en este caso sabemos que el array solo va a tener un elemento, por lo tanto a la plantilla le asignamos la **canciones[0]** (primer y único elemento del array).

```
app.get('/cancion/:id', function (req, res) {
  var criterio = { "_id" : req.params.id };

  gestorBD.obtenerCanciones(criterio, function(canciones){
    if ( canciones == null ){
      res.send(respuesta);
    } else {
      var respuesta = swig.renderFile('views/bcancion.html',
      {
        cancion : canciones[0]
      });
      res.send(respuesta);
    }
  });
});
```

Aunque aparentemente el código este bien, si guardamos los cambios y ejecutamos la aplicación observamos que nunca se retorna ninguna canción. Esto se debe a que estamos tratando la **id** como un "String" ( { "\_id" : req.params.id } ), cuando realmente es un objeto, debemos transformar el String en un **ObjectID**, el módulo de mongo cuenta con una función específica para esa tarea **ObjectID()**.

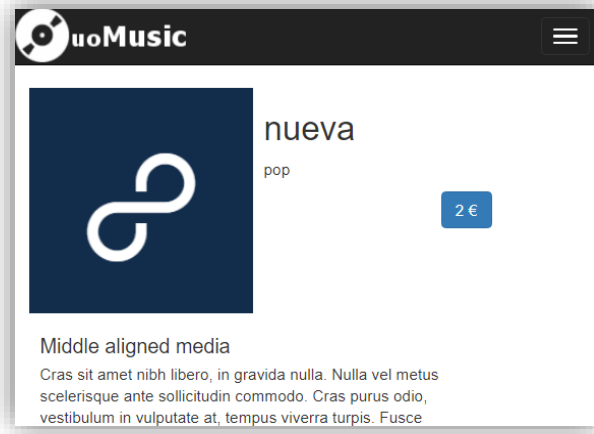
```
app.get('/cancion/:id', function (req, res) {
  var criterio = { "_id" : gestorBD.mongo.ObjectID(req.params.id) };

  gestorBD.obtenerCanciones(criterio, function(canciones){
    if ( canciones == null ){
```





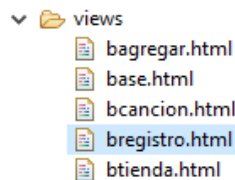
Guardamos los cambios ejecutamos la aplicación y comprobamos que la vista de detalles se muestra correctamente.



## Registro de usuarios

Vamos a incluir un formulario para registrar usuarios, el registro consistiría en almacenar un usuario en una colección llamada “**usuarios**”. El conjunto de datos mínimo que podríamos guardar de un usuario pueden ser: “**\_id (automatico)**”, “**email**”, “**password**”.

Copiamos la vista **/public/bregistro.html** a **/views/bregistro.html**



Queremos que al recibir la petición **GET /registrarse** se muestre la vista de **bregistro.html** (esta vista no recibe ningún parámetro). Como se trata de una vista relativa a los usuarios la implementamos en el controlador **/routes/usuarios.js**

```
app.get("/registrarse", function(req, res) {  
  var respuesta = swig.renderFile('views/bregistro.html', {});  
  res.send(respuesta);  
});
```

El formulario contenido en **bregistro.html** envía una petición **POST /usuario** con los parámetros **nombre** y **email**. Deberíamos almacenar un usuario con esos datos.

Antes de continuar vamos a añadir el “require” del módulo **Crypto** <https://nodejs.org/api/crypto.html> (se trata de un módulo incluido en el core no requiere descarga), el cual vamos a utilizar para encriptar los password en lugar de guardarlos en texto plano. Añadimos el módulo en el fichero principal **app.js**



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

var crypto = require('crypto');
```

Vamos a agregar dos variables nuevas a la aplicación una será la **clave de cifrado**, y otra la referencia al propio **módulo crypto** (en lugar de pasarlo como parámetro a los controladores, como hicimos con swig y mongo podemos incluirlo como variable de aplicación, esta es otra vía para poder utilizarlo en los controladores).

```
// Variables
app.set('port', 8081);
app.set('db', 'mongodb://localhost:27017/uomusic');
app.set('clave', 'abcdefg');
app.set('crypto', crypto);
```

Volvemos a **usuarios.js** e implementamos la respuesta a **/POST usuario**.

1. Obtenemos el parámetro **req.body.password** y lo encriptamos con **crypto**
2. Creamos un objeto usuario incluyendo el password seguro.

```
app.post('/usuario', function(req, res) {

    var seguro = app.get("crypto").createHmac('sha256', app.get('clave'))
        .update(req.body.password).digest('hex');

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

})
```

Ahora solo falta insertar el usuario en la base de datos, para ello implementamos la función **insertarUsuario()** en el **gestorBD.js**.

```
module.exports = {
    mongo : null,
    app : null,
    init : function(app, mongo) {
        this.mongo = mongo;
        this.app = app;
    },
    insertarUsuario : function(usuario, functionCallback) {
        this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {
            if (err) {
                functionCallback(null);
            } else {
                var collection = db.collection('usuarios');
                collection.insert(usuario, function(err, result) {
                    if (err) {
                        functionCallback(null);
                    } else {
                        functionCallback(result.ops[0]._id);
                    }
                });
            }
        });
    }
}
```



```
db.close();
});
}
});
},
```

Ejecutamos la función **insertarUsuario** desde **POST /usuario**

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

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

  gestorBD.insertarUsuario(usuario, function(id) {
    if (id == null){
      res.send("Error al insertar ");
    } else {
      res.send('Usuario Insertado ' + id);
    }
  });
});
})
```

Guardamos los cambios. Accedemos a <http://localhost:8081/registrarse> y comprobamos que funciona correctamente. Creamos el usuario [prueba1@prueba1.com](mailto:prueba1@prueba1.com) (password: prueba1) y el usuario [prueba2@prueba2.com](mailto:prueba2@prueba2.com) (password: prueba2).

Registrar usuario

Email:  
prueba@prueba.com

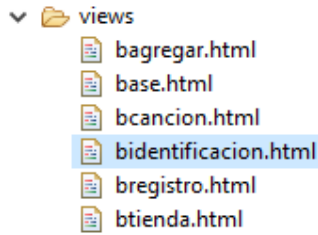
Password:  
.....

Registrar

**NOTA:** no vamos a implementarlo, pero antes de insertar el usuario habría que comprobar si la colección usuarios cuenta ya con un usuario con el mismo Email, en ese caso deberíamos notificarle al usuario que el email ha sido usado.

## Identificación de usuario

Copiamos la vista **/public/bidentificación.html** en el directorio **views**.



En el fichero **bidentificación.html** hay definido un formulario con una acción **POST /identificarse** que envía los parámetros **email** y **password**.

```
{% block contenido_principal %}
<h2>Identificación de usuario</h2>
<form class="form-horizontal" method="post" action="/identificarse">
  <div class="form-group">
```

Accedemos al controlador **usuarios.js** e implementamos la respuesta a **GET /identificarse**.

```
app.get("/identificarse", function(req, res) {
  var respuesta = swig.renderFile('views/bidentificacion.html', {});
  res.send(respuesta);
});
```

El formulario presente en la vista envía una petición **POST /identificarse** con los parámetros **email** y **password**. En la lógica de la petición comprobamos si existe un usuario con ese email y password (encriptado) realizando un **find()**. Si el **find()** devuelve un error o 0 resultados significa que los datos no son correctos.

Vamos a implementar una nueva función **obtenerUsuarios()** en el **gestorBD.js**, el proceso va a ser el mismo que para **obtenerCanciones()** incluiremos un **criterio** configurable.

```
module.exports = {
  mongo : null,
  app : null,
  init : function(app, mongo) {
    this.mongo = mongo;
    this.app = app;
  },
  obtenerUsuarios : function(criterio,funcionCallback){
    this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {
      if (err) {
        funcionCallback(null);
      } else {
        var collection = db.collection('usuarios');
        collection.find(criterio).toArray(function(err, usuarios) {
          if (err) {
            funcionCallback(null);
          } else {
            funcionCallback(usuarios);
          }
        });
        db.close();
      }
    });
  }
};
```



Por el momento únicamente respondemos si hemos conseguido identificar al usuario, si retorna 0 coincidencias no hemos conseguido identificarlo.

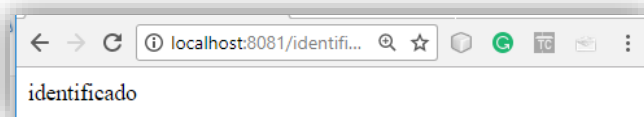
Para finalizar, la función **POST /identificarse** obtendrá los datos del usuario y realizará una búsqueda a través de **gestorBD.obtenerUsuarios(criterio)**, en caso de que la función retorne coincidencias se trata de un usuario identificado. Debemos recordar que para que la búsqueda sea correcta el password del usuario debe estar encriptado siguiendo el mismo sistema que utilizamos cuando se salvaron los datos del usuario.

```
app.post("/identificarse", 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(usuario) {
    if (usuario == null || usuario.length == 0) {
      res.send("No identificado: ");
    } else {
      res.send("identificado");
    }
  });
});
```

Guardamos los cambios, ejecutamos la aplicación y accedemos a <http://localhost:8081/identificarse> para comprobar que el funcionamiento es correcto debería permitir identificarnos si introducimos los datos de un usuario ya registrado.



## Uso de sesión

Vamos a utilizar el módulo **express-session**. Para instalarlo abrimos la consola cmd, y nos situamos el directorio del proyecto, ejecutamos el comando **npm install express-session --save**

```
C:\Users\jordanosy\work\TiendaMusica>cd C:\Users\jordanosy\work\TiendaMusica
C:\Users\jordanosy\work\TiendaMusica>npm install express-session --save
TiendaMusica@0.1.0 C:\Users\jordanosy\work\TiendaMusica
`-- express-session@1.15.6
   |-- crc@3.4.4
   |-- on-headers@1.0.1
   |-- uid-safe@2.1.5
   `-- random-bytes@1.0.0
```

Abrimos el fichero principal **app.js** y declaramos el módulo **express-session**, podemos configurar algunos aspectos de la sesión, como el secreto que se va a utilizar para codificar los



identificadores de sesión, también podemos especificar si la información de la sesión puede ser modificada o su inicialización. <https://github.com/expressjs/session>.

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

var expressSession = require('express-session');
app.use(expressSession({
  secret: 'abcdefg',
  resave: true,
  saveUninitialized: true
}));
var crypto = require('crypto');
```

A partir de este momento ya podemos utilizar la sesión, se accede a ella con **req.session.<clave\_del\_objeto>**, podemos guardar objetos nuevo bajo cualquier clave que elijamos, modificar objetos existentes o consultarlos. La sesión se utiliza de forma muy recurrente en las aplicaciones web para guardar información del usuario, por ejemplo: si está identificado en el sitio o las compras que hay en su carrito.

Modificamos la lógica de la respuesta **POST /identificarse** para guardar el **email** (identificador único) del usuario en sesión cuando este se identifica correctamente.

```
app.post("/identificarse", 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) {
      req.session.usuario = null;
      res.send("No identificado: ");
    } else {
      req.session.usuario = usuarios[0].email;
      res.send("identificado");
    }
  });
});
```

También incluimos una respuesta a **GET /desconectarse** para eliminar el usuario de sesión.

```
app.get('/desconectarse', function (req, res) {
  req.session.usuario = null;
  res.send("Usuario desconectado");
});
```

Utilizando la variable **req.session.usuario** podemos ver si hay un usuario identificando. Vamos a realizar dos modificaciones en el proceso de agregar una canción **POST /canción** del controlador **rcanciones.js**, solo usuarios identificados podrán agregar canciones.



1. **POST /cancion** (agregar canción) comprueba si hay usuario identificado en sesión (**req.session.usuario**), si no lo hay nos redirige a **/tienda**
2. **POST /cancion** (agregar canción) además de los datos introducidos por el usuario el objeto canción va a tener un **autor**, con el valor **req.session.usuario** (email del usuario)

```
app.post("/cancion", function(req, res) {  
  if ( req.session.usuario == null){  
    res.redirect("/tienda");  
    return;  
  }  
  
  var cancion = {  
    nombre : req.body.nombre,  
    genero : req.body.genero,  
    precio : req.body.precio,  
    autor: req.session.usuario  
  }  
})
```

Podríamos comprobar si hay un usuario identificado en la petición **GET /cancion/agregar**

```
app.get('/canciones/agregar', function (req, res) {  
  if ( req.session.usuario == null){  
    res.redirect("/tienda");  
    return;  
  }  
  
  var respuesta = swig.renderFile('views/bagregar.html', {});  
  res.send(respuesta);  
})
```

Guardamos los cambios, ejecutamos la aplicación y comprobamos que la funcionalidad sea correcta.

- Primero nos aseguramos de que no hay usuario en sesión <http://localhost:8081/desconectarse>
- Intentamos acceder a <http://localhost:8081/canciones/agregar> sin usuario en sesión, nos debería dejar que nos envíe a **/tienda**.
- Nos identificamos en <http://localhost:8081/identificarse>
- Agregamos una canción <http://localhost:8081/canciones/agregar>
- Buscamos la canción en la tienda <http://localhost:8081/tienda> y verificamos que tiene un **autor**



## Colecciones relacionadas – Usuario y canciones

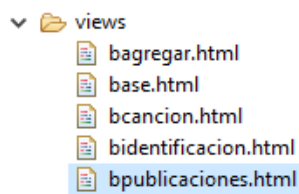
Vamos a implementar que **GET /publicaciones** retorne únicamente las canciones publicadas por el autor actualmente identificado en sesión (ver las canciones que he publicado). El código va a ser igual al de **GET / tienda**, pero incluyendo un criterio { autor : req.session.usuario };

```
app.get("/publicaciones", function(req, res) {
    var criterio = { autor : req.session.usuario };

    gestorBD.obtenerCanciones(criterio, function(canciones) {
        if (canciones == null) {
            res.send("Error al listar ");
        } else {
            var respuesta = swig.renderFile('views/btienda.html',
            {
                canciones : canciones
            });
            res.send(respuesta);
        }
    });
});
```

Si nos identificamos con un usuario y accedemos a <http://localhost:8081/publicaciones> veremos que únicamente aparecen las publicaciones de las cuales es autor.

En lugar de reutilizar la vista **btienda.html** vamos a usar **bppublicaciones.html** muestra información de las canciones, pero en otro formato. Movemos la vista de la carpeta **/public/** a la carpeta **/views/**

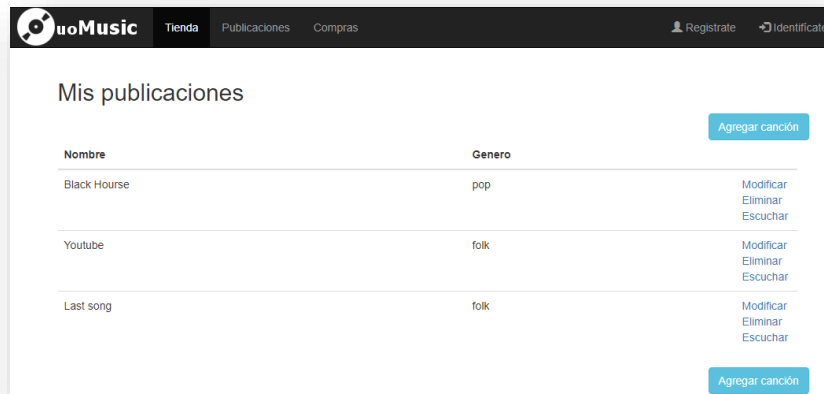


Cambiamos el nombre de la vista en **GET /publicaciones**





```
var respuesta = swig.renderFile('views/bpublicaciones.html',
```



## Control de acceso por enrutador

Comprobar en todas las funciones del controlador si el usuario está identificado no suele ser una buena estrategia. Habíamos seguido esa estrategia en **GET /cancion/agregar** y **POST /cancion/agregar**.

```
app.get('/canciones/agregar', function (req, res) {  
  if ( req.session.usuario == null){  
    res.redirect("/tienda");  
    return;  
  }  
})
```

Este tipo de implementaciones son difíciles de mantener y solo nos sirve para controlar el acceso a peticiones declaradas en el controlador, por ejemplo, deja sin comprobar los accesos a los recursos de **/audios/** si alguien obtiene la URL de un mp3 podría reproducirlo, aunque no fuese una de sus canciones.

Declaramos una variable **routerUsuarioSession** que será un router (**express.Router()**). Un router puede **interceptar** las peticiones que van dirigidas a otras URLs, analizarlas realizar redirecciones o dejarlas correr.

Diseño del control de acceso, lógica del router:

1. Si hay un usuario en sesión dejamos correr las peticiones.
2. Si no hay usuario en sesión, guardamos la dirección a donde se dirigía la petición (**req.originalUrl**) y redireccionamos a **GET /identificarte**

Una vez declarado el router se debe especificar sobre que URLs se aplica, utilizando **app.use(ruta, router)**. Debemos especificar todas las rutas sobre las que se va a aplicar (al añadir una ruta captura todo tipo de peticiones GET, POST, etc.). En este caso lo aplicaremos sobre **/audios/** (archivos contenidos en public/audios), **/publicaciones** y **/canciones/agregar**.



Incluimos el enrutador en el fichero principal de la aplicación **app.js** . Es importante agregar el router a la aplicación en la zona correcta

1. Después del módulo “express-session” ya que dentro del router vamos a utilizar la **sesión**.
2. Antes de declarar el directorio **public** como estático, ya que si lo declaramos después de haber declarado el directorio estático estaríamos indicando que el directorio estático tiene prioridad.
3. Antes que los controladores de **usuarios y canciones**, ya que si lo declaramos después estaríamos indicando que la gestión de los controladores tiene prioridad.

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

var expressSession = require('express-session');
app.use(expressSession({
  secret: 'abcdefg',
  resave: true,
  saveUninitialized: true
}));
var crypto = require('crypto');
var fileUpload = require('express-fileupload');
app.use(fileUpload());
var mongo = require('mongodb');
var swig = require('swig');
var bodyParser = require('body-parser');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

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

// routerUsuarioSession
var routerUsuarioSession = express.Router();
routerUsuarioSession.use(function(req, res, next) {
  console.log("routerUsuarioSession");
  if ( req.session.usuario ) {
    // dejamos correr la petición
    next();
  } else {
    console.log("va a : "+req.session.destino)
    res.redirect("/identificarse");
  }
});

//Aplicar routerUsuarioSession
app.use("/canciones/agregar",routerUsuarioSession);
app.use("/publicaciones",routerUsuarioSession);
app.use("/audios/",routerUsuarioSession);

app.use(express.static('public'));
```

Ahora podríamos eliminar las condiciones que comprobaban si había un usuario en sesión en GET /canciones/agregar y POST /cancion.

Guardamos los cambios, ejecutamos la aplicación y comprobamos que el funcionamiento es correcto. Sin identificarnos previamente tratamos de acceder a las rutas protegidas



**/publicaciones, /canciones/agregar** o a un fichero contenido en **/audios**, nos debería dirigir a /identificarse.

**NOTA:** No obstante, sigue habiendo un problema que pasa inadvertido en algunos sitios web. En los recursos **/audio/** comprobamos que el usuario está en sesión, pero no que ese usuario tiene permiso para acceder al fichero de la canción, por ejemplo, solo debería tener acceso si es el autor o si la ha comprado.

Eliminamos la ruta **/audio/** de los “use” de **routerUsuarioSession**, creamos un nuevo **router routerAudios**, el cual obtenga de la URL de la petición el nombre del fichero mp3 (coincide con la ID de la canción) <http://localhost:8081/audios/59949e0c42f1152dacdca2f6.mp3>

El módulo de Node **path** nos ofrece muchos métodos de soporte para extraer partes de rutas <https://nodejs.org/api/path.html> en este caso lo usamos para obtener la ID de la canción, luego recuperamos la canción con esa ID y comprobamos si el autor de les el usuario que hay en sesión (**req.session.usuario**). Dejamos avanzar la petición si el usuario coincide sino lo enviamos a **/tienda**.

```
//Aplicar routerUsuarioSession
app.use("/canciones/agregar",routerUsuarioSession);
app.use("/publicaciones",routerUsuarioSession);
app.use("/audios/",routerUsuarioSession);

//routerAudios
var routerAudios = express.Router();
routerAudios.use(function(req, res, next) {
  console.log("routerAudios");
  var path = require('path');
  var idCancion = path.basename(req.originalUrl, '.mp3');

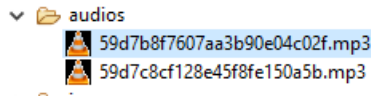
  gestorBD.obtenerCanciones(
    { _id : mongo.ObjectID(idCancion) }, function (canciones) {
      if( canciones[0].autor == req.session.usuario ){
        next();
      } else {
        res.redirect("/tienda");
      }
    })
});

//Aplicar routerAudios
app.use("/audios/",routerAudios);

app.use(express.static('public'));
```

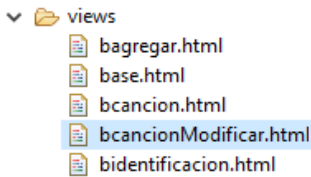


Comprobamos el funcionamiento, estando identificados, pero intentando acceder a un fichero de audio del cual el usuario no sea autor. Podemos ver las ids desde los ficheros del proyecto:



## Modificar canciones

Movemos la vista **bcancionModificar.html** del directorio **/public** al **/views**.



**bcancionModificar.html** es una combinación de las vistas utilizadas en las peticiones:

- **/cancion/:id** -> recibe como parámetro un objeto canción y muestra sus datos, pero está vez sobre un formulario (para que puedan ser editados).
- **/cancion/publicar** muestra un formulario muy similar al de agregar canción (mismos campos con las mismas claves), pero lo envía a **POST /cancion/modificar/:id**

```
<h2>Modificar canción</h2>
<form class="form-horizontal" method="post" action="/cancion/modificar/{{ cancion.id.toString() }}">
  <div class="form-group">
```

Implementamos en el controlador la respuesta al **GET /cancion/modificar/:id** obtenemos la canción con la id que se recibe como parámetro y se carga la vista **bcancionModificar.html**

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

  gestorBD.obtenerCanciones(criterio, function(canciones){
    if ( canciones == null ){
      res.send(respuesta);
    } else {
      var respuesta = swig.renderFile('views/bcancionModificar.html',
      {
        cancion : canciones[0]
      });
      res.send(respuesta);
    }
  });
});
```

Antes de implementar la respuesta a **POST /cancion/modificar/:id** nos dirigimos al **gestorBD.js** e implementamos la función **modificarCancion()**. Para actualizar un documento de una colección utilizamos **collection.update()** esta función recibe dos parámetros:



1. Criterios para seleccionar la canción a modificar, seguramente será tener una id determinada
2. Canción: las propiedades de la canción que quieran ser restablecidas, pueden ser todas o algunas, incluso nuevas propiedades que serán agregadas dinámicamente al objeto.

El **result** de la actualización es un objeto que contiene las propiedades que han sido modificadas, no contiene todas las propiedades solo las modificadas, no tendrá id

```
},
modificarCancion : function(criterio, cancion, funcionCallback) {
    this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {
        if (err) {
            funcionCallback(null);
        } else {
            var collection = db.collection('canciones');
            collection.update(criterio, {$set: cancion}, function(err, result) {
                if (err) {
                    funcionCallback(null);
                } else {
                    funcionCallback(result);
                }
            });
            db.close();
        }
    });
},
});
},
```

Abrimos el controlador **rcanciones.js** e implementamos la respuesta a **POST /cancion/modificar/:id**, hay que obtener la **id** y después crear un objeto canción con los nuevos valores.

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

    var cancion = {
        nombre : req.body.nombre,
        genero : req.body.genero,
        precio : req.body.precio
    }

    gestorBD.modificarCancion(criterio, cancion, function(result) {
        if (result == null) {
            res.send("Error al modificar ");
        } else {
            res.send("Modificado "+result);
        }
    });
});
});
```

La función está casi completa, pero falta comprobar si la petición contiene ficheros **req.files.portada**, **req.files.audio**. Si estos ficheros se almacenaran en la BD este proceso sería muy sencillo, los agregaríamos en la propia consultar.

Al tratarse de ficheros que se deben subir al servidor y de modificación opcional debemos seguir una secuencia.



- (Paso 1) Intentamos subir la **portada**
  - A. Si se produce un error al subir la portada enviamos una respuesta de error
  - B. Sí se sube correctamente vamos al (Paso 2) Intentamos subir el **audio**
  - C. Si no había portada vamos al (Paso 2)
- (Paso 2) Intentamos subir el **audio**
  - A. Si se produce un error al subir el audio enviamos una respuesta de error
  - B. Sí se sube correctamente vamos al (Final)
  - C. Si no había portada vamos al (Final)

Aunque podríamos implementar toda la lógica en la función **POST /cancion/modificar/:id** el código quedará más claro y fácil de modificar si sacamos los pasos a otras funciones.

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

  var cancion = {
    nombre : req.body.nombre,
    genero : req.body.genero,
    precio : req.body.precio
  }

  gestorBD.modificarCancion(criterio, cancion, function(result) {
    if (result == null) {
      res.send("Error al modificar ");
    } else {
      res.send("Modificado-" + result);
      paso1ModificarPortada(req.files, id, function (result) {
        if( result == null){
          res.send("Error en la modificación");
        } else {
          res.send("Modificado");
        }
      });
    }
  });
});

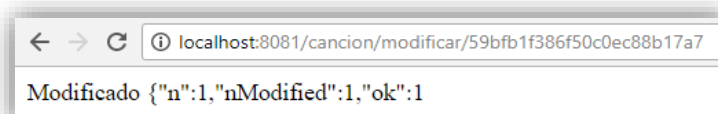
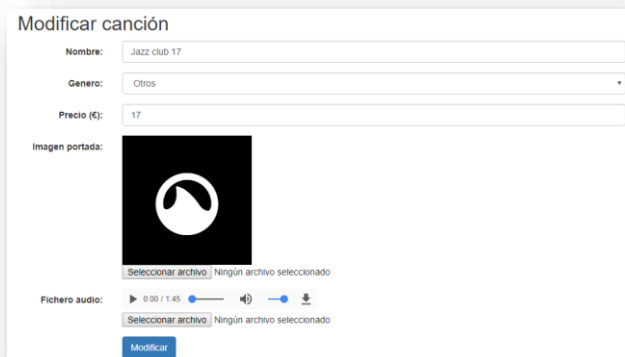
function paso1ModificarPortada(files, id, callback){
  if (files.portada != null) {
    var imagen = files.portada;
    imagen.mv('public/portadas/' + id + '.png', function(err) {
      if (err) {
        callback(null); // ERROR
      } else {
        paso2ModificarAudio(files, id, callback); // SIGUIENTE
      }
    });
  } else {
    paso2ModificarAudio(files, id, callback); // SIGUIENTE
  }
}

function paso2ModificarAudio(files, id, callback){
  if (files.audio != null) {
    var audio = files.audio;
    audio.mv('public/audios/' + id + '.mp3', function(err) {
      if (err) {
        callback(null); // ERROR
      } else {
        callback(true); // FIN
      }
    });
  }
}
```

```
}  
    });  
  } else {  
    callback(true); // FIN  
  }  
}
```

Es importante que eliminemos la respuesta **res.send()** enviada antes de la subida de imágenes. Puesto que no podemos seguir procesando una petición una vez ha sido enviada la respuesta.

Si ejecutamos la aplicación podemos comprobar que la funcionalidad parece correcta. Al identificarnos y acceder a <http://localhost:8081/publicaciones> podemos modificar una de nuestras canciones.



## Propuestas de repaso

- Crea una nueva aplicación Node.js express
- Permite registrar e identificar usuarios: nombre de usuario y contraseña
- Comprueba que no se puedan registrar dos usuarios con el mismo nombre, el nombre actuará al igual que la `_id` como clave única.
- Permite que los usuarios registrados puedan crear comentarios, estos comentarios se almacenarán en una base de datos mongo en la colección "comentarios". Un comentario debe tener: nombre del autor (obtenido automáticamente del usuario en sesión), texto y fecha (generada automáticamente).
- Mostrar todos los comentarios almacenados en la lista
- Permitir que solo usuarios identificados puedan acceder a la lista de comentarios
- Permitir que un usuario pueda modificar un comentario si este había sido escrito por él.

