



Sistemas Distribuidos e Internet

Desarrollo de aplicaciones Web con NodeJS

Sesión - 8

Curso 2019/ 2020



Contenido

1	Introducción.....	3
2	Acceso a datos de Mongo desde Node.js	3
3	Arquitectura para el acceso a datos.....	7
4	Subida de ficheros y gestión de colecciones.....	10
4.1	Recuperar y listar las canciones	13
4.2	Listar canciones por criterio	15
4.3	Vista de detalles de la canción	17
5	Registro de usuarios.....	20
6	Identificación de usuario.....	23
7	Uso de sesión	25
8	Colecciones relacionadas – Usuarios y canciones.....	28
9	Control de acceso por enrutador	29
10	Modificar canciones.....	32
11	Resultado esperado en el repositorio de GitHub	36



1 Introducción

A lo largo de esta práctica, continuaremos con el desarrollo de aplicaciones Web ágiles con NodeJS. Concretamente, conectaremos a bases de datos no relaciones (MongoDB) e implementaremos un sistema de autenticación de usuarios, sesiones y enrutadores.

2 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/>). Es uno de los múltiples módulos externos existentes para conectarse fácilmente a mongo en NodeJS.

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

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

¡Importante! Instalaremos la versión 2.*. Puesto que tiene nomenclatura diferente a la 3.0

```
c:\Dev\workspaces\NodeApps\sdi-lab-node>npm install mongodb@2.2.33
+ mongodb@2.2.33
updated 1 package and audited 181 packages in 3.524s
found 1 low severity vulnerability
  run `npm audit fix` to fix them, or `npm audit` for details
```

Para poder acceder al módulo mongodb incluimos la sentencia **require('mongodb')** y almacenaremos el objeto que nos devuelve en **app.js**. Este objeto, lo enviaremos como parámetro al controlador **rcanciones** (igual que hicimos con swig). Por último, creamos una variable con clave **set.db('db', <valor>)** en la que introducimos la **cadena de conexión**

Recomendado: Usa la cadena de conexión obtenida (cada uno la suya) al final del guion anterior de prácticas a través de <https://cloud.mongodb.com>. Ejemplo de conexión:

```
mongodb://admin:<password>@tiendamusica-shard-00-00-hy8gh.mongodb.net:27017,
tiendamusica-shard-00-01-hy8gh.mongodb.net:27017,tiendamusica-shard-00-02-
hy8gh.mongodb.net:27017/test?ssl=true&replicaSet=tiendamusica-shard-
0&authSource=admin&retryWrites=true
```

Alternativa: Si tuviéramos MongoDB instalado en local: **mongodb://localhost:27017/uomusic**

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

let mongo = require('mongodb');
let swig = require('swig');
let 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://admin:<password>@tiendamusica-shard-00-00-hy8gh.mongodb.net:27017');

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

Recordatorio: SUSTITUYE LA CADENA DE CONEXIÓN DEL EJEMPLO POR LA PROPIA.



Modificamos el fichero **rcanciones.js** para que reciba el parámetro **mongo**:

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

La función que procesa la petición para agregar canciones a través del formulario, ya está implementada (**POST /cancion**).

La función que procesa la petición enviada por el formulario ya está implementada (es **POST /canción**). A continuación, creamos un **objeto** canción con los parámetros recibidos a través del formulario. Por ahora, el código de la función quedaría así:

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

Una vez definido el objeto, tenemos que conectarnos a la base de datos para almacenarlo. Usaremos la función **connect()** del objeto **MongoClient** que requiere estos parámetros:

1. La **cadena de conexión**: la tenemos almacenada en la variable de aplicación 'db'.
2. La **función(err, db) 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 ejecutan las acciones (insertar, consultar, etc.).

En el cuerpo de la función que enviamos como parámetro, incluiremos el código para insertar el objeto **canción** en la base de datos. Lo primero, será recuperar la colección de canciones con **db.collection(canciones)** para seguidamente, insertar la canción mediante la función **insert**. La función **insert** recibe como parámetros:

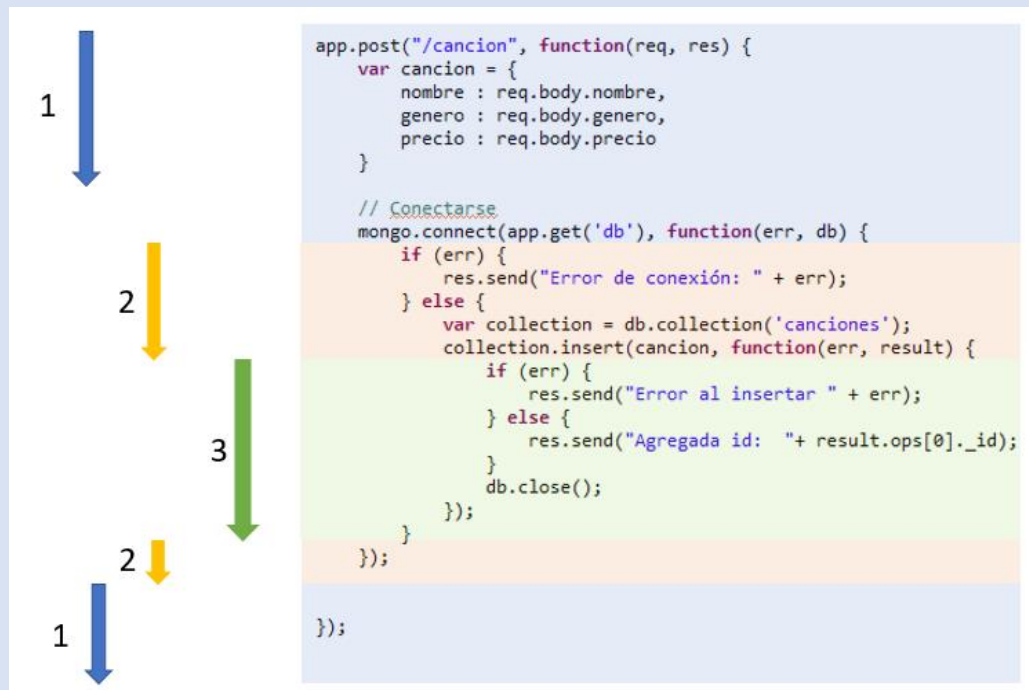
1. El propio objeto a insertar: el objeto **cancion**
2. Una **función(err, result)** que se ejecuta al finalizar la operación insert. Esta función la utilizaremos para cerrar la conexión a la base de datos con **db.close()** y enviar una respuesta al cliente. A continuación, se explican los parámetros de la función:
 - **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/objeto y queremos consultar su **_id** debemos acceder a **result.ops[0]._id**.



Como se puede apreciar en el siguiente código, la respuesta que enviaremos al cliente dependerá de si se ha insertado con éxito el objeto o no.

```
app.post("/cancion", function(req, res) {
  let 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 {
      let 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 NodeJS) **se ejecuta de forma asíncrona** mediante un sistema de callbacks.

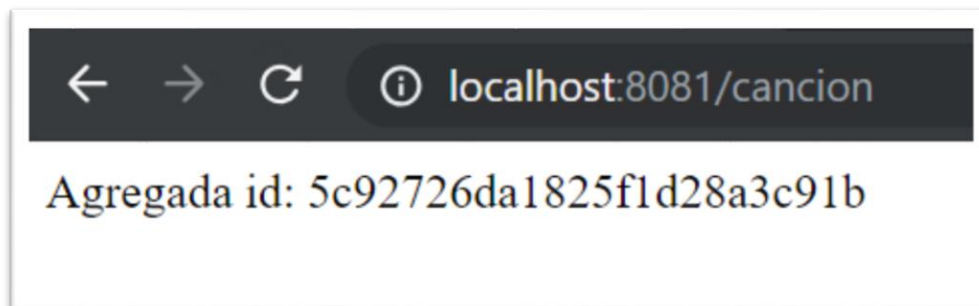




Ejecutamos el `app.js` e intentamos agregar una canción para comprobar que el código funciona correctamente. Accedemos a <http://localhost:8081/canciones/agregar> y completamos el formulario.

Nota: Debemos asegurarnos de que la función `app.get("canciones/:id")` **no está** declarada antes que `app.get("canciones/agregar")`. Si no, entraremos en la función incorrecta empleando el valor "agregar" para el parámetro `id`.

Una vez agregada la respuesta será del tipo:



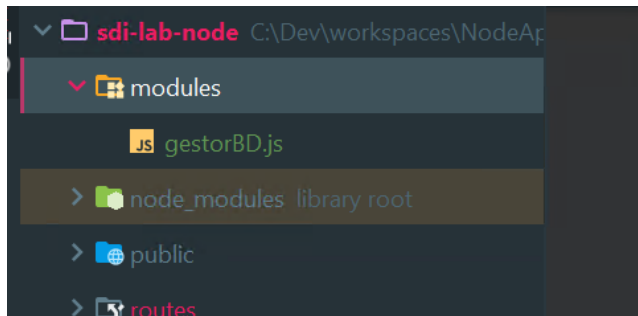
Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-8.2 - Acceso a datos de Mongo desde Node.js"*.



3 Arquitectura para el acceso a datos

La implementación anterior, a pesar de ser funcional, 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.

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



A diferencia de los módulos anteriores **rcanciones** y **rusuarios**, este módulo es un **OBJETO**, con variables y funciones a las que nosotros decidimos cuándo 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 usar this para acceder a las variables desde dentro del objeto.**

Copiamos el siguiente código en el fichero gestorBD.js:

```
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 {
        let collection = db.collection('canciones');
        collection.insert(cancion, function(err, result) {
          if (err) {
            functionCallback(null);
          } else {
            functionCallback(result.ops[0]._id);
          }
        });
        db.close();
      }
    });
  }
};
```



En este objeto hemos definido:

- Dos variables: **app** y **mongo**.
- Una función **init(app,mongo)** para inicializar las dos variables globales.
- Una función **insertarCancion** **asíncrona**, **por lo que no puede tener return**. **Para devolver un valor, lo pasa como parámetro a la función de callback**. Posibles valores:
 - 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**.

Para utilizar este objeto tenemos que agregarlo en **app.js** con un **require**, y llamar a su función **init(app,mongo)**. **La variable mongo debe inicializarse ANTES de llamar a gestorDB.init**.

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

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

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

A continuación, vamos a enviar el nuevo objeto **gestorBD** a los controladores. Como **gestorBD** **ya incluye el objeto mongo**, no es necesario que se lo pasemos a los controladores.

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

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

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

Y de igual manera en **rcanciones**.

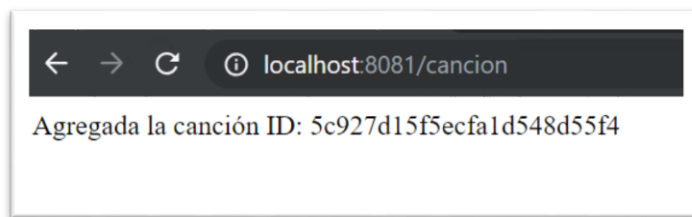
```
module.exports = function(app, swig, gestorBD) {
```




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

```
app.post("/cancion", function(req, res) {  
  let 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 canción");  
    } else {  
      res.send("Agregada la canción ID: " + id);  
    }  
  });  
});
```

Agregamos una canción y comprobamos que todo sigue funcionando correctamente. Accedemos a <http://localhost:8081/canciones/agregar> y completamos el formulario.



Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-8.3 - Arquitectura para el acceso a datos"*.



4 Subida de ficheros y gestión de colecciones

Vamos a implementar la lógica asociada a la subida de la imagen y el fichero de audio empleando 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:\Dev\workspaces\NodeApps\sdi-lab-node>npm install express-fileupload
+ express-fileupload@1.1.3-alpha.1
added 10 packages from 4 contributors and audited 195 packages in 3.079s
found 1 low severity vulnerability
  run `npm audit fix` to fix them, or `npm audit` for details
```

Añadimos el módulo en el fichero **app.js**. En este caso, **no es necesario enviarlo como parámetro a los controladores puesto que se integra en la aplicación mediante app.use.**

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

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

Modificamos la petición **POST /canciones**, puesto que una vez este guardada la canción en la base de datos no enviaremos directamente la respuesta. La respuesta se enviará una vez que se complete la subida de los ficheros.

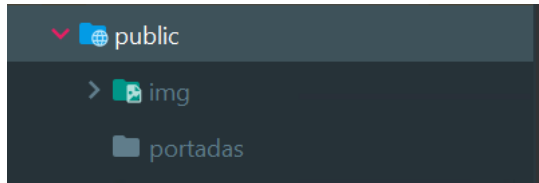
Comprobamos si en la request ha llegado un fichero con el nombre portada (**req.files.portada.name**). Si es así, guardamos el fichero en una variable y lo salvamos con la función **mv(path, función de callback)**. Una vez se ejecute la función de callback, el fichero habrá sido almacenado. Dependiendo 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("Agregado id: " + id);
    if (req.files.portada != null) {
      let 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.**



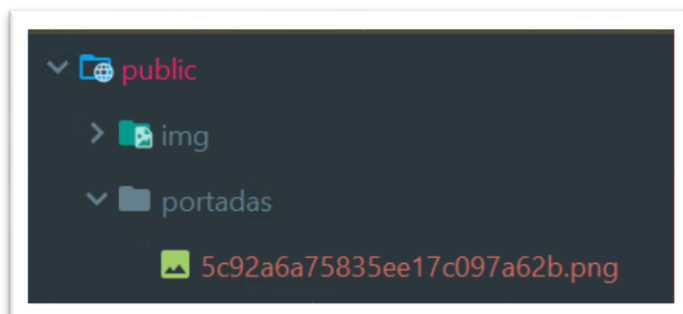
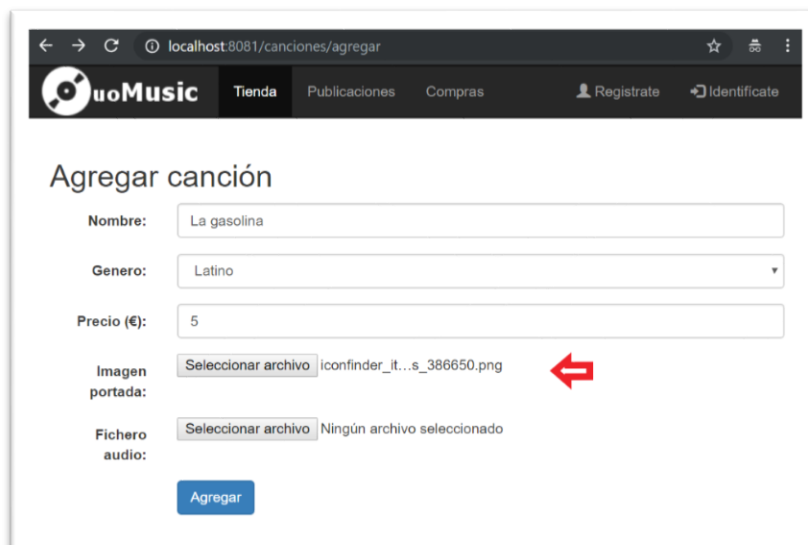
Como hemos indicado que el fichero de portada se guarda en la carpeta **public/portadas**, debemos crearla o el proceso fallará.



Antes de probarlo debemos asegurarnos de que el formulario contiene la propiedad **encType="multipart/form-data"** con el fin de que pueda procesar ficheros. 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, probando a agregar una nueva canción con una imagen como portada. El fichero de portada debería aparecernos ahora en la carpeta **public/portadas**.





Para subir el fichero de audio el proceso es muy similar. Una vez se ha subido con éxito la portada, comprobaremos si la petición contiene `req.files.audio.name`.

```
if (req.files.portada != null) {
  let imagen = req.files.portada;
  imagen.mv('public/portadas/'+cancionId+'.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) {
        let 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` en el que se almacenarán los audios subidos.



Como último paso, modificamos el formulario de la vista `bagregar.html` para incluir que los ficheros sean obligatorios (**required**) y 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, ejecutar la aplicación y probamos a añadir una canción. Dentro del fichero **canciones.rar** disponible en el campus virtual hay varios recursos (imágenes y audios) que podemos utilizar para crear canciones.

4.1 Recuperar y listar las canciones

Vamos a añadir el código necesario para mostrar el catálogo de todas las canciones almacenadas en la base de datos a través de la petición **GET /tienda**. Abrimos el fichero **gestorBD.js** y agregamos una nueva función **obtenerCanciones()**.

La función **obtenerCanciones** va a retornar todos los documentos almacenados en la **colección** "canciones" a través de la función **.find()**. Si no establecemos un criterio a **find()**, 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 {
        let 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** y realizamos una llamada a **obtenerCanciones()**. Una vez recibida la lista de la base de datos, la enviamos como parámetro a la vista **btienda.html** usando la clave **canciones**.

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



Si guardamos los cambios y ejecutamos la aplicación podremos ver en <http://localhost:8081/tienda> el catálogo de canciones. Siempre y cuando previamente las hayamos guardado desde nuestra aplicación (a través del formulario agregar).

La vista todavía 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 contenido 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`, es necesario obtener el String con la función `toString()`.

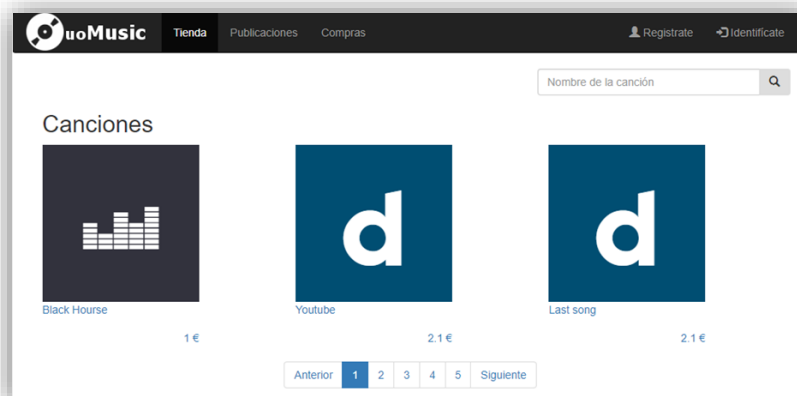
Ejemplo de documentos Mongo, `_id` es de tipo `ObjectId`, no es una cadena:

```
> 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" }
```

El código a añadir en `btienda.html` sería el siguiente:

```
{% 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>
  </div>
</div>
{% endfor %}
```

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



Nota: Las canciones agregadas previamente a este apartado, aparecerán sin imagen y sin audio. Es posible gestionar los documentos desde <https://cloud.mongodb.com/>. Si navegamos a cluster → Collections, será posible eliminar/modificar los elementos.



4.2 Listar canciones por criterio

Vamos a introducir un nuevo parámetro en la función **obtenerCanciones()** del **gestorBD.js**. Este parámetro será el criterio que deben de cumplir las canciones para ser retornadas, similar a la condición where. Este criterio, será utilizado como parámetro en la función **find(<criterio>)**. En el caso de no necesitar un criterio, enviaríamos un objeto vacío {}.

Por ejemplo, si quisiéramos obtener la canción con nombre "Black House" el criterio sería:

```
{ "nombre": "Black House" }
```

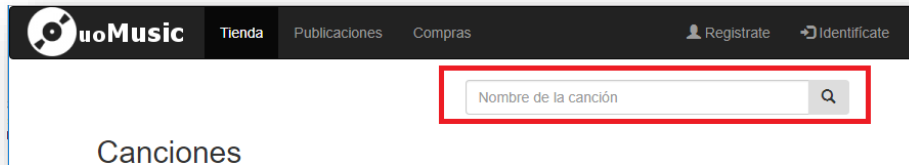
```
obtenerCanciones : function(criterio,functionCallback){
  this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {
    if (err) {
      functionCallback(null);
    } else {
      let collection = db.collection('canciones');
      collection.find(criterio).toArray(function(err, canciones) {
        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()** será el del objeto vacío.

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

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

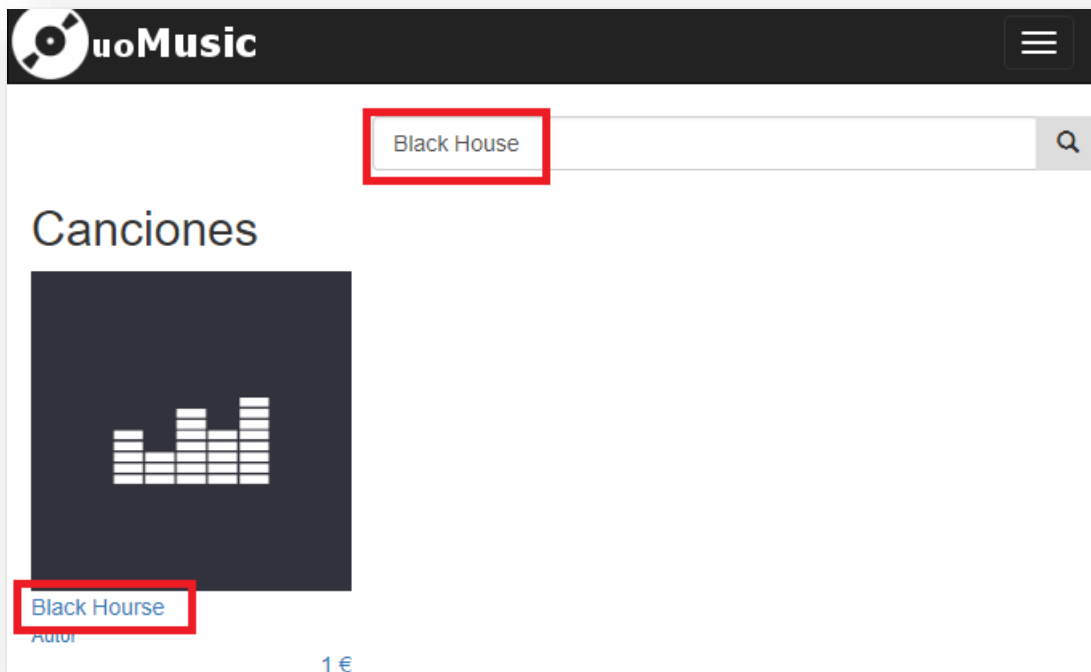
La búsqueda de la aplicación está pensada para buscar canciones por nombre. Este formulario envía una petición **GET /tienda?busqueda=<nombre>**. Debemos obtener el **parámetro busqueda** con el objetivo final de mostrar en la vista únicamente las canciones que coincidan con ese criterio.



En el controlador **GET /tienda** comprobamos si la petición tiene el parámetro **req.query.busqueda**. Si no hay parámetro, el criterio de búsqueda será el objeto vacío **{ }** (todas las canciones). Si hay parámetro, el criterio será: **{ "nombre" : req.query.busqueda }**

```
app.get("/tienda", function(req, res) {  
  let criterio = {};  
  if( req.query.busqueda != null ){  
    criterio = { "nombre" : req.query.busqueda };  
  }  
}
```

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/>

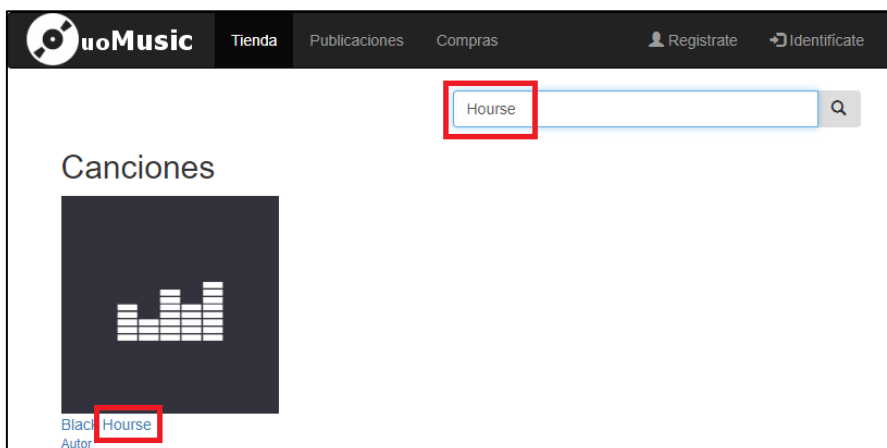
También expresiones regulares/patrones:

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

En lugar realizar la búsqueda por nombre exacto, vamos a modificar el criterio para seleccionar **cualquier canción que contenga el texto de búsqueda** en su nombre.

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

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



4.3 Vista de detalles de la canción

En la tienda, al pulsar sobre la miniatura de una canción queremos que nos muestre el detalle de la misma. A continuación, vamos a implementar esta funcionalidad.

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

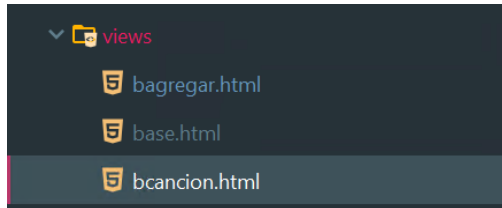
```
{% for cancion in canciones %}  
<div class="col-xs-6 col-sm-6 col-md-4 col-lg-3">  
  <div style="width: 200px">  
    <a href="/cancion/{{ cancion._id.toString() }}">  
        
      <!-- http://www.socicon.com/generator.php -->  
      <div>{{ cancion.nombre }}</div>  
      <div class="small">{{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

Movemos el fichero **public/bcancion.html** que habíamos descargado del Campus virtual (material de la sesión anterior) a la carpeta **/views**.



El fichero proporciona la estructura HTML y necesitamos incluir las sentencias swig para mostrar los valores de los atributos de la canción. Si un parámetro de la canción no existe, como **canción.autor**, simplemente no se mostrará sin producir error.

```
{% 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>
```

Por último, implementamos la respuesta **GET /cancion/{id}**. La funcionalidad consistirá en ejecutar la función **obtenerCanciones()** para poner la canción a disposición de la vista **bcancion.html**. El criterio a utilizar en la función **obtenerCanciones()** será la **_id** de la canción.



Hay que tener en cuenta que la función devolverá un array. Sin embargo, sabemos que solo va a tener un elemento, por lo tanto a la plantilla le enviamos **canciones[0]** (primer y único elemento del array).

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

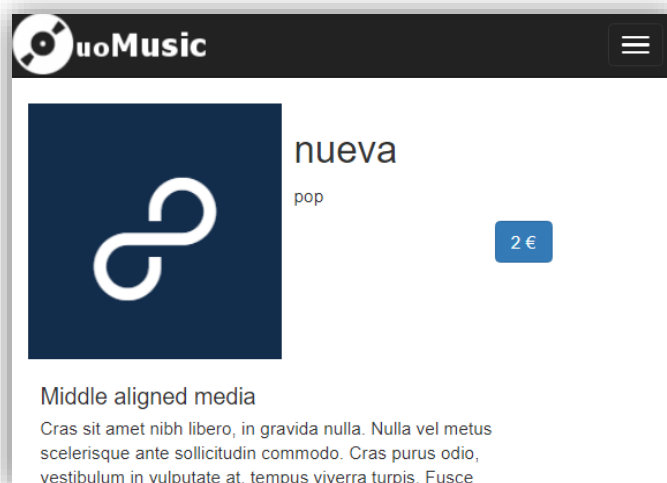
  gestorBD.obtenerCanciones(criterio, function(canciones){
    if ( canciones == null ){
      res.send(respuesta);
    } else {
      let respuesta = swig.renderFile('views/bcancion.html',
        {
          cancion : canciones[0]
        });
      res.send(respuesta);
    }
  });
});
```

Aunque aparentemente el código esté bien, si guardamos los cambios y ejecutamos la aplicación observamos que **nunca se retorna ninguna canción**. Esto es **debido a que estamos empleando la _id como String cuando debemos tratarla como un objeto (ObjectID)**. Utilizamos la función `ObjectID()` de mongo para transformar el texto en objeto:

```
app.get('/cancion/:id', function (req, res) {
  let 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.



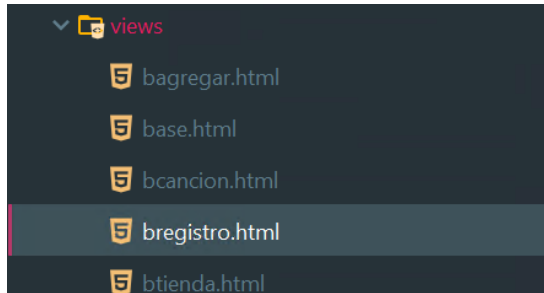
Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-8.4-Subida de ficheros y gestión de colecciones"*.



5 Registro de usuarios

En este apartado, incluiremos un formulario para registrar usuarios. El registro consistirá en almacenar a los usuarios en una colección llamada “**usuarios**”. Almacenaremos los siguientes datos del usuario: **_id** (automático), **email** y **password**.

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



Queremos que, al recibir la petición **GET /registrarse**, se muestre la vista de **bregistro.html**. Como se trata de una vista relativa a los usuarios, la implementamos en el controlador **/routes/rusuarios.js**

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

El formulario incluido en **bregistro.html** envía una petición **POST /usuario** con los parámetros **nombre** y **email**. Por tanto, debemos crear y almacenar un usuario con esos datos.

Antes de continuar, vamos a añadir un módulo para encriptar el password para no almacenarlo como texto plano. El módulo **Crypto** (<https://nodejs.org/api/crypto.html>) **ya viene incluido con NodeJS, por lo que no necesitamos descargarlo**. Añadimos el módulo en el fichero principal **app.js** mediante un require.

```
// Módulos  
let express = require('express');  
let app = express();  
  
let crypto = require('crypto');
```

Configuramos el módulo añadiendo una **clave de cifrado** y una **referencia al módulo Crypto**. La referencia a crypto, la almacenaremos como variable dentro de app. Empleando este mecanismo, no será necesario enviarlo a cada uno de los controladores (ya enviamos app).

```
// 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 **crypt**.
2. Creamos un objeto usuario incluyendo el password seguro.

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

Implementamos la función **insertarUsuario()** en el **gestorBD.js** para almacenar al usuario:

```
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 {  
        let collection = db.collection('usuarios');  
        collection.insert(usuario, function(err, result) {  
          if (err) {  
            functionCallback(null);  
          } else {  
            functionCallback(result.ops[0]._id);  
          }  
        });  
        db.close();  
      }  
    });  
  }  
};
```

Añadimos la ejecución de la función **insertarUsuario** en **POST /usuario**:

```
app.post('/usuario', function(req, res) {  
  let seguro = app.get("crypto").createHmac('sha256', app.get('clave'))  
    .update(req.body.password).digest('hex');  
  
  let usuario = {  
    email : req.body.email,  
    password : seguro  
  }  
  
  gestorBD.insertarUsuario(usuario, function(id) {  
    if (id == null){  
      res.send("Error al insertar el usuario");  
    } else {  
      res.send('Usuario Insertado ' + id);  
    }  
  });  
  
})
```



Guardamos los cambios, ejecutamos la aplicación y accedemos a <http://localhost:8081/registrarse>. Vamos comprobar el funcionamiento, incluyendo dos usuarios:

- Usuario prueba1@prueba1.com (password: prueba1)
- Usuario prueba2@prueba2.com (password: prueba2).

Registrar usuario

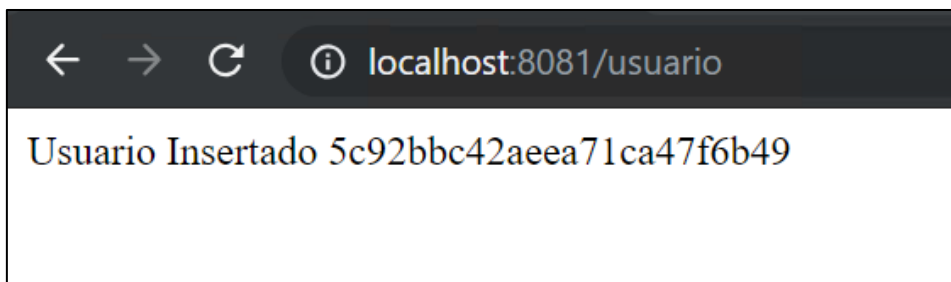
Email:

prueba@prueba.com

Password:

.....

Registrar

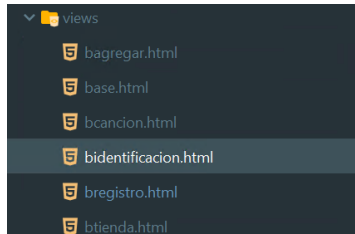


Nota: Antes de insertar el usuario habría que comprobar si la colección usuarios cuenta ya con un usuario con el mismo email. En ese caso, deberíamos notificarle al usuario que el email ya está en uso. Todavía no lo implementaremos.



6 Identificación de usuario

En este apartado, añadiremos el proceso de autenticación de usuarios. Como primer paso, movemos la vista **/public/bidentificacion.html** a la carpeta **views**.



En el fichero **bidentificacion.html** hay definido un formulario **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 **rusuarios.js** e implementamos la respuesta a **GET /identificarse**.

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

El formulario anterior envía una petición **POST /identificarse** con los parámetros **email** y **password**. Por tanto, comprobaremos si existe un usuario con ese email y password (encriptado) realizando un **find()**. Si devuelve un error o 0 resultados, los datos son erróneos. Vamos a implementar una nueva función **obtenerUsuarios()** en el **gestorBD.js**. El proceso será ser el mismo que para **obtenerCanciones()**, incluyendo 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 {
        let collection = db.collection('usuarios');
        collection.find(criterio).toArray(function(err, usuarios) {
          if (err) {
            funcionCallback(null);
          } else {
            funcionCallback(usuarios);
          }
        });
        db.close();
      }
    });
  }
};
```



Para finalizar, la función **POST /identificarse** recibirá los datos del usuario (enviados mediante el formulario) y realizará una búsqueda a través de **gestorBD.obtenerUsuarios(criterio)**. Si la consulta devuelve resultado(s), el proceso de autenticación ha sido realizado correctamente.

```
app.post("/identificarse", function(req, res) {  
  let seguro = app.get("crypto").createHmac('sha256', app.get('clave'))  
    .update(req.body.password).digest('hex');  
  
  let criterio = {  
    email : req.body.email,  
    password : seguro  
  }  
  
  gestorBD.obtenerUsuarios(criterio, function(usuarios) {  
    if (usuarios == null || usuarios.length == 0) {  
      res.send("No identificado: ");  
    } else {  
      res.send("identificado");  
    }  
  });  
});
```

Debemos fijarnos en que **el password se encripta antes de enviarlo** porque si no, no coincidirá con el almacenado (el almacenado también fue encriptado antes de insertarlo).

Para finalizar, guardamos los cambios, ejecutamos la aplicación y accedemos a <http://localhost:8081/identificarse>. Para comprobar que el funcionamiento es correcto, **debería permitir identificarnos con alguno de los usuarios previamente registrados**.



7 Uso de sesión

Para gestionar la sesión, utilizaremos el módulo **express-session**. Para instalarlo abrimos la consola cmd, nos situamos en el directorio del proyecto y ejecutamos el comando **npm install express-session --save**

```
c:\Dev\workspaces\NodeApps\sdi-lab-node>npm install express-session --save
+ express-session@1.15.6
added 5 packages from 5 contributors and audited 207 packages in 2.454s
found 1 low severity vulnerability
run `npm audit fix` to fix them, or `npm audit` for details
```

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. Detalles en: <https://github.com/expressjs/session#express-session>

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

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

A partir de este momento ya podemos utilizar la sesión. **Para acceder al objeto sesión emplearemos: req.session.<clave_del_objeto>**. En este objeto, podemos: almacenar otros objetos bajo cualquier clave que elijamos, modificar/acceder a objetos existentes.

Modificamos la lógica de **POST /identificarse** para que, cuando el usuario se identifique correctamente, almacenar en sesión su **email** (porque es un identificador único).

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

  let 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");  
})
```

A través de la variable **req.session.usuario** podemos saber si el usuario está autenticado. En base a lo anterior, vamos a modificar en **rcanciones.js** el proceso de agregar canciones (**POST /cancion**), para que **solo usuarios identificados puedan agregar canciones**.

1. **POST /cancion** (agregar canción) comprueba si hay usuario identificado en sesión (**req.session.usuario**). Si no lo hay, 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;  
  }  
  
  let cancion = {  
    nombre : req.body.nombre,  
    genero : req.body.genero,  
    precio : req.body.precio,  
    autor: req.session.usuario  
  }  
})
```

También debemos comprobar si hay un usuario identificado en **GET /cancion/agregar**

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

Modificamos la vista **btienda.html** para que muestre el nombre del autor.

```
<a href="/cancion/{{ cancion._id.toString() }}">   
<!-- http://www.socicon.com/generator.php -->  
<div class="wrap">{{ cancion.nombre }}</div>  
<div class="small">Autor: {{ cancion.autor }} </div>  
<div class="text-right">{{ cancion.precio }} €</div>  
</a>
```



Guardamos los cambios, ejecutamos la aplicación y comprobamos que la funcionalidad está implementada correctamente:.

- 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 y nos debería redirigir a **/tienda** (no hemos iniciado sesión).
- 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**



Nota: Subir el código a GitHub en este punto. Commit Message → *“SDI-NodeJS-8.5,67- Registro de usuario, autenticación y uso de sesión”*.



8 Colecciones relacionadas – Usuarios y canciones

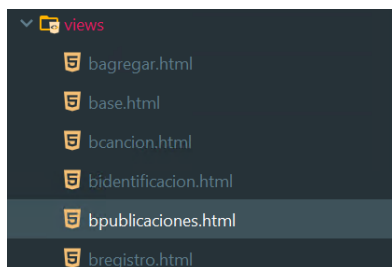
En este apartado, vamos a implementar una respuesta (**GET /publicaciones**) que retorne únicamente las canciones publicadas por el autor (usuario) actualmente identificado en sesión. El código será como el de **GET /tienda**, pero con la particularidad de que necesitamos un filtro { autor : req.session.usuario };

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

  gestorBD.obtenerCanciones(criterio, function(canciones) {
    if (canciones == null) {
      res.send("Error al listar ");
    } else {
      let 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 canciones en las que figura como autor.

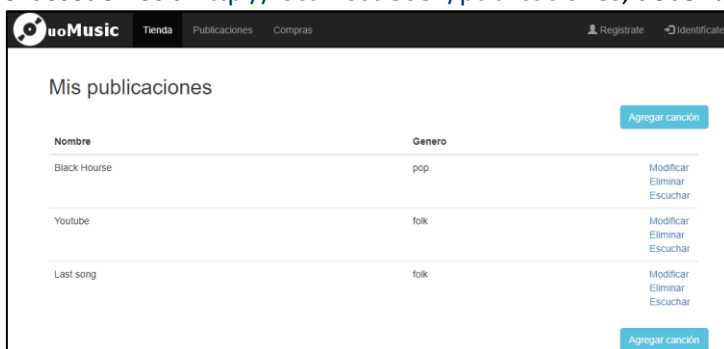
En lugar de reutilizar la vista **btienda.html** vamos a usar **bpublicaciones.html**. Esta vista muestra información de las canciones, pero en otro formato. Movemos la vista de la carpeta **/public/** a la carpeta **/views/**



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

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

Si accedemos a <http://localhost:8081/publicaciones>, deberíamos ver la nueva vista tal que así:





9 Control de acceso por enrutador

Comprobar en todas las funciones del controlador si el usuario está identificado no suele ser una buena estrategia. Hasta ahora, hemos 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 poner un ejemplo, deja sin comprobar los accesos a los recursos de **/audios/**. Por tanto, si alguien obtiene la URL de un .mp3 podría reproducirlo, aunque no fuese una de sus canciones.

Vamos a crear un router (**express.Router()**) que **intercepte** todas las peticiones dirigidas a URLs. Una vez interceptada la petición, la analizaremos y decidiremos si la dejamos continuar o redirigimos a otra URL. La lógica para decidir qué hacer, se basará en:

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 redirigimos a **GET /identificarse**

Una vez declarado el router, debemos especificar sobre qué URLs se aplica mediante **app.use(ruta, router)**. Al especificar rutas, tenemos que tener en cuenta que captura cualquier tipo de petición (GET, POST, etcétera). En este caso lo aplicaremos sobre: **/audios/** (ficheros 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 posición 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 de los controladores de usuarios y canciones**, ya que si lo declaramos después **estaríamos indicando que la gestión de los controladores tiene prioridad**.

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



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

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

// routerUsuarioSession
let 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 /canción, ya que pasa a ser un código redundante.

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**. El resultado esperado es que nos redirija 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` y creamos un nuevo `router routerAudios`. Este nuevo router, obtendrá de la URL el nombre del fichero mp3 (que coincide con la ID de la canción). Ejemplo: `http://localhost:8081/audios/59949ef1152dacdca2f6.mp3`

El módulo de Node `path` (<https://nodejs.org/api/path.html>) nos ofrece muchos métodos de soporte para extraer partes de ruta. En este caso, lo usamos para obtener la ID de la canción y recuperar la canción que tenga esa ID. Una vez tengamos la canción, comprobamos si el autor de la canción es el usuario que hay en sesión (`req.session.usuario`). Dejamos avanzar la petición si el usuario coincide, en caso contrario lo enviamos a `/tienda`.

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

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

  gestorBD.obtenerCanciones(
    { "_id": mongo.ObjectID(idCancion) }, function (canciones) {

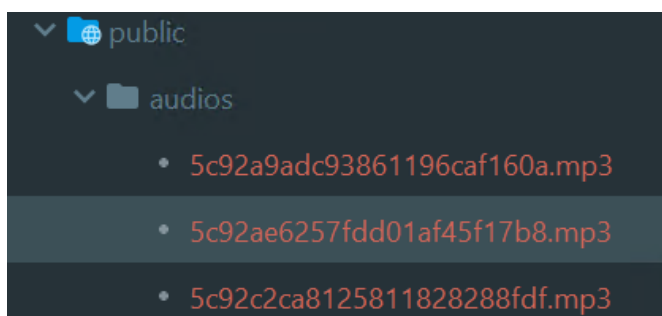
      if(req.session.usuario && canciones[0].autor == req.session.usuario ){
        next();
      } else {
        res.redirect("/tienda");
      }
    })
});

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

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

¡OJO! En este punto, debéis **tener cuidado con las canciones que no tengan autor, creadas en apartados anteriores**. El código `canciones[0].autor` podría lanzar una excepción, por lo que es aconsejable que vaciéis la BD y la rellenéis con un par de canciones especificando un autor.

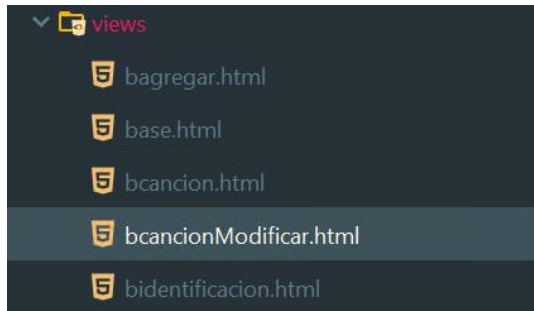
Guardamos los cambios, ejecutamos y comprobamos el funcionamiento. Estando identificados, intentamos acceder a un audio del que no somos propietarios. Podemos ver las ids desde los ficheros del proyecto:





10 Modificar canciones

En este último apartado, vamos a incluir la funcionalidad necesaria para poder modificar las canciones creadas. Como primer paso, tenemos la vista **bcancionModificar.html** de la carpeta **/public** a la carpeta **/views**.



bcancionModificar.html es una combinación de las vistas utilizadas en las peticiones **/canciones/agregar**. El cambio fundamental, es que el formulario aparecerá con los valores de la canción a modificar y 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 a **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) {
  let criterio = { "_id" : gestorBD.mongo.ObjectId(req.params.id) };

  gestorBD.obtenerCanciones(criterio, function(canciones){
    if (canciones == null){
      res.send(respuesta);
    } else {
      let 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()**, que recibe dos parámetros:

1. Criterios para seleccionar la canción a modificar. Por ejemplo: que tenga una **_id** concreta.
2. Canción: un objeto canción con las propiedades (y nuevos valores) que va a ser modificada. Las propiedades pueden modificarse todas, algunas o incluso añadir nuevas serán agregadas al objeto de la base de datos.



El **result** de la actualización es un objeto que contiene las propiedades que han sido modificadas.

¡Solamente contiene las modificadas por lo que no tendrá `_id`!

```
},
modificarCancion : function(criterio, cancion, funcionCallback) {
    this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {
        if (err) {
            funcionCallback(null);
        } else {
            let 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**. Obtenemos la `_id` a partir del parámetro `:id` y creamos un objeto canción con los nuevos valores.

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

    let 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 los ficheros **req.files.portada** y **req.files.audio**. Si estos ficheros se almacenaran en la BD este proceso sería muy sencillo porque los agregaríamos en la propia consulta. Al tratarse de ficheros que se deben subir al servidor y **de modificación opcional** debemos seguir una secuencia:

1. Paso I: 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 a Paso II e intentamos subir el **audio**
 - C. Si no había portada vamos a Paso II



2. Paso II: Intentamos subir el **audio**

- A. Si se produce un error al subir el audio enviamos una respuesta de error
- B. Si se sube correctamente, finalizamos.
- C. Si no había audio, finalizamos.

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

  let 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 && files.portada != null) {
    let 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 && files.audio != null) {
    let 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.


Modificar canción

Nombre:

Genero:

Precio (€):

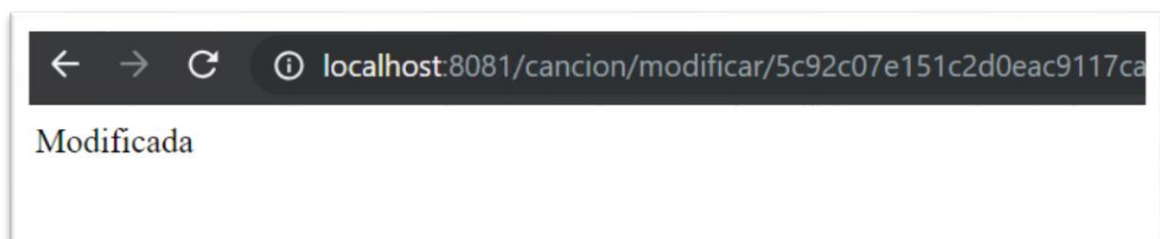
Imagen portada:



Ningún archivo seleccionado

Fichero audio:

Ningún archivo seleccionado



Nota: Subir el código a GitHub en este punto. Commit Message -> *“SDI-NodeJS-8.9 y 8.10 - Colecciones relacionadas y routers”*.



11 Resultado esperado en el repositorio de GitHub

Al revisar el repositorio de código en GitHub, los resultados de los commits realizados deberían ser parecidos a estos.

Commits on Mar 15, 2020

- SDI-NodeJS-8.9 y 8.10 - Colecciones relacionadas y routers**
melsanchezsantillan committed 3 minutes ago
- SDI-NodeJS-8.5,67- Registro de usuario, autenticación y uso de sesión**
melsanchezsantillan committed 2 hours ago
- SDI-NodeJS-8.4-Subida de ficheros y gestión de colecciones**
melsanchezsantillan committed 4 hours ago
- SDI-NodeJS-8.3 - Arquitectura para el acceso a datos**
melsanchezsantillan committed 7 hours ago
- SDI-NodeJS-8.2 - Acceso a datos de Mongo desde Node.js**
melsanchezsantillan committed 20 hours ago