



Sistemas Distribuidos e Internet

Curso 2022/2023

Desarrollo de aplicaciones web con Node.js

Sesión - 8



Contenido

1	Introducción	3
2	Acceso a datos de Mongo desde Node.js	3
3	Arquitectura para el acceso a datos	6
4	Subida de ficheros y gestión de colecciones	9
4.1	Recuperar y listar las canciones	13
4.2	Listar canciones por criterio y otras opciones	15
4.3	Vista de detalles de una canción	19
5	Registro de usuarios	22
6	Autenticación de usuario	26
7	Uso de sesión	29
8	Colecciones relacionadas – Usuarios y canciones	32
9	Control de acceso por enrutador	34
10	Editar canciones	39
11	Etiquetar el proyecto	44
12	Resultado esperado en el repositorio de GitHub	45



!!!!!!MUY IMPORTANTE!!!!!!

Aunque en este guion se ha usado como nombre de repositorio para todo el ejercicio **sdiIDGIT-lab-nodejs**, cada alumno deberá usar como nombre de repositorio **sdi**x**-lab-nodejs**, donde **x** = columna **IDGIT** en el Documento de ListadoAlumnosSDI2223.pdf del CV.

En resumen, por ejemplo, para el alumno **IDGIT=2223-101**:

Repositorio remoto:	usar el mismo repositorio que en el guion anterior.
Repositorio local:	usar el mismo repositorio que en el guion anterior.
Nombre proyecto:	trabajaremos sobre el proyecto del guion anterior.
Colaborador invitado:	sdigithubuniovi

1 Introducción

A lo largo de esta práctica, continuaremos con el desarrollo de aplicaciones web ágiles con Node.js. 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 desde Node.js.

Abrimos la consola de comandos y nos situamos en el directorio raíz del proyecto. Ejecutamos **npm install mongodb@4.1.4**, (última versión del módulo de mongodb 4.1.X)

```
PS C:\Dev\Projects\SDI_projects\sdiIDGIT-lab-nodejs\musicstoreapp> npm install mongodb@4.1.2
up to date, audited 94 packages in 918ms

5 packages are looking for funding
  run `npm fund` for details

1 high severity vulnerability

To address all issues, run:
  npm audit fix

Run `npm audit` for details.
```

Para corregir el problema de vulnerabilidad de la aplicación ejecutamos el siguiente comando: **npm audit fix**. Así, actualizará los paquetes que presentan problemas de seguridad

```
PS C:\Dev\Projects\SDI_projects\sdiIDGIT-lab-nodejs\musicstoreapp> npm audit fix
removed 1 package, changed 1 package, and audited 93 packages in 3s

5 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```



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

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>. Esta cadena de conexión también puede guardarse en un fichero de configuración. Además, hay que cambiar **`<password>`** por la contraseña generada en mongo cloud. Ejemplo de conexión:

```
const { MongoClient } = require("mongodb");
const url = 'mongodb+srv://admin:<password>@eii-sdi-
cluster.xjf0khu.mongodb.net/?retryWrites=true&w=majority';
app.set('connectionStrings', url);
require("./routes/songs.js")(app, MongoClient);
```

Recuerda: SUSTITUYE LA CADENA DE CONEXIÓN DEL EJEMPLO POR LA TUYA.

A continuación, modificamos el módulo **`songs.js`** para que reciba el parámetro **`mongoClient`**:

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

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

```
app.post('/songs/add', function (req, res) {
  let song = {
    title: req.body.title,
    kind: req.body.kind,
    price: req.body.price
  }
});
```

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

1. La **cadena de conexión**: la tenemos almacenada en la variable de aplicación `'connectionStrings'`.
2. La **function(err, dbClient)** que se ejecutará al completar la conexión. Esta función tiene dos parámetros:
 - o **err**: es un objeto donde se almacenan los errores, es nulo/vacío si no ha habido ningún problema.



- **dbClient**: es una referencia al cliente de la base de datos. Sobre este objeto se ejecutan las acciones (insertar, consultar, borrar, etc.).

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

1. El propio objeto a insertar: el objeto **song**
2. Devuelve una promesa al ejecutar la operación insert. La promesa devolverá los siguientes resultados:
 - **result**: Retorna un documento (objeto) si la promesa se ha cumplido, conteniendo:
 - Un campo **insertedId** con el valor **_id** del documento insertado.
 - Un booleano **acknowledged** indica si la operación de inserción fue exitosa o no.
 - **err**: es un objeto donde se almacenan los errores si la promesa ha sido rechazada. Es nulo/vacío si no ha habido ningún problema.

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('/songs/add', function (req, res) {
  let song = {
    title: req.body.title,
    kind: req.body.kind,
    price: req.body.price
  }
  MongoClient.connect(app.get('connectionStrings'), function (err, dbClient) {
    if (err) {
      res.send("Error de conexión: " + err);
    } else {
      const database = dbClient.db("musicStore");
      const collectionName = 'songs';
      const songsCollection = database.collection(collectionName);
      songsCollection.insertOne(song)
        .then(result => res.send("canción añadida id: " + result.insertedId))
        .then(() => dbClient.close())
        .catch(err => res.send("Error al insertar " + err));
    }
  });
});
```

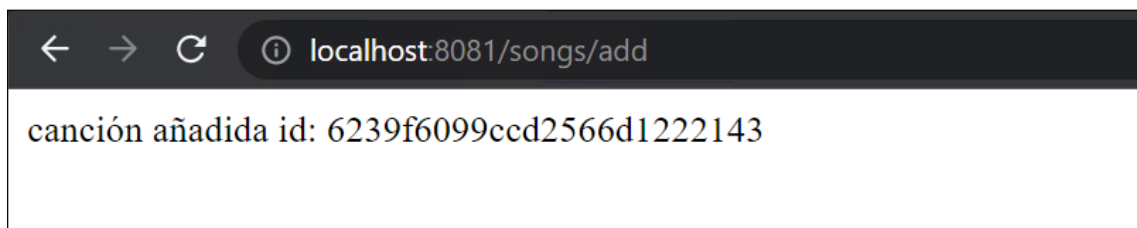


¡Importante! Debemos tener en cuenta que el código de **MongoClient.connect** (como el de muchas funciones en JavaScript y Node.js) **se ejecuta de forma asíncrona** mediante un sistema de callbacks y/o promesas.

Ejecutamos el proyecto (www.js) e intentamos agregar una canción para comprobar que el código funciona correctamente. Accedemos a <http://localhost:8081/songs/add> y completamos el formulario.

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

Una vez añadida la canción, la respuesta será del tipo:



Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-8.1-Acceso a datos de Mongo desde Node.js."

OJO: sustituir IDGIT por tu número asignado (p.e. 2223-101):

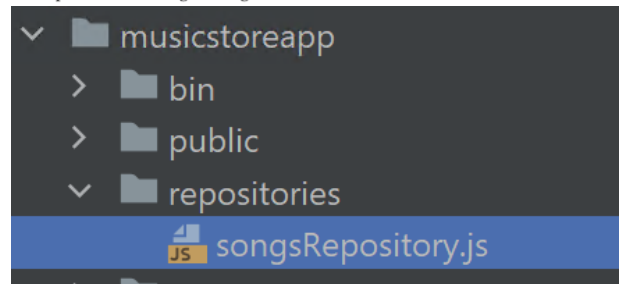
"SDI-2223-101-8.1-Acceso a datos con MongoDB desde Node.js."

(No olvides incluir los guiones y **NO** incluyas BLANCOS al principio o al final, ni las comillas)

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 mover la gestión de la base de datos a un **Módulo-objeto** independiente.

Creamos una nueva carpeta **repositories** en el directorio raíz y creamos dentro un fichero **songsRepository.js**.



A diferencia de los módulos anteriores (**songs** y **users**), este módulo es un **OBJETO** con variables y funciones a las que nosotros decidimos cuándo llamar. El módulo songs es 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 JavaScript.

Diferencias:

- Las variables globales se declaran usando **clave: 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 el propio objeto.**

Copiamos el siguiente código en el fichero songsRepository.js:

```
module.exports = {  
  mongoClient: null,  
  app: null,  
  init: function (app, mongoClient) {  
    this.mongoClient= mongoClient;  
    this.app = app;  
  },  
  insertSong: function (song, callbackFunction) {  
  
    this.mongoClient.connect(this.app.get('connectionStrings'), function (err, dbClient) {  
      if (err) {  
        callbackFunction(null)  
      } else {  
        const database = dbClient.db("musicStore");  
        const collectionName = 'songs';  
        const songsCollection = database.collection(collectionName);  
        songsCollection.insertOne(song)  
          .then(result => callbackFunction(result.insertedId))  
          .then(() => dbClient.close())  
          .catch(err => callbackFunction({error: err.message}));  
      }  
    });  
  }  
};
```

En este objeto hemos definido:



- Dos variables: **app** y **mongoClient**.
- Una función **init(app, mongoClient)** para inicializar las dos variables globales.
- Una función **insertSong** **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 **objeto con el mensaje del error**.

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

```
...
app.set('connectionStrings', url);
let songsRepository = require("./repositories/songsRepository.js");
songsRepository.init(app, MongoClient);
require("./routes/songs.js")(app, songsRepository);
require("./routes/songs.js")(app, MongoClient);
...
```

A continuación, vamos a enviar el nuevo objeto **songsRepository** a los controladores. Como **songsRepository** ya incluye el objeto **mongoClient**, no es necesario que se lo pasemos a los controladores. **Obviamente, los repositorios deben estar definidos ANTES que los controladores**.

```
require("./routes/songs.js")(app, songsRepository);
//require("./routes/songs.js")(app, MongoClient);
```

Indicamos que este parámetro se va a recibir en el router **songs**

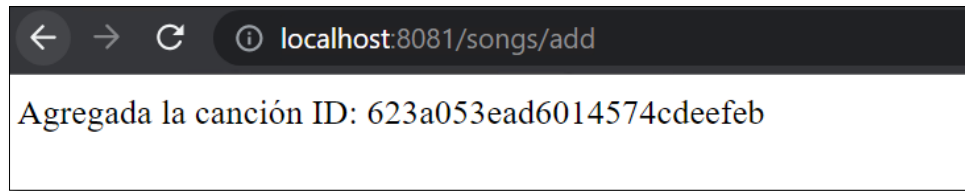
```
module.exports = function (app, songsRepository) {..}
```

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

```
app.post('/songs/add', function (req, res) {
  let song = {
    title: req.body.title,
    kind: req.body.kind,
    price: req.body.price
  }
  songsRepository.insertSong(song, function (songId) {
    if (songId == null) {
      res.send("Error al insertar canción");
    } else {
      res.send("Agregada la canción ID: " + songId);
    }
  });
});
```




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



Nota: Incluir el siguiente Commit Message ->

“SDI-IDGIT-8.2-Arquitectura para el acceso a datos.”

OJO: sustituir IDGIT por tu número asignado (p.e. 2223-101):

“SDI-2223-101-8.2-Arquitectura para el acceso a datos.”

(No olvides incluir los guiones y NO incluyas BLANCOS al principio o al final, ni las comillas)

4 Subida de ficheros y gestión de colecciones

Vamos a implementar la lógica asociada a la subida de imágenes y de ficheros de audio empleando el módulo externo **express-fileupload**¹. Accedemos por consola de comandos al directorio raíz del proyecto y ejecutamos el comando **npm install express-fileupload --save**

```
PS C:\Dev\Projects\SDI_projects\sdiIDGIT-lab-nodejs\musicstoreapp> npm install express-fileupload
added 4 packages, and audited 97 packages in 4s

5 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Dev\Projects\SDI_projects\sdiIDGIT-lab-nodejs\musicstoreapp>
```

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**. Además, añadimos una variable en la app para establecer la ruta base donde almacenaremos las portadas y los audios.

```
...
let app = express();
let fileUpload = require('express-fileupload');
app.use(fileUpload({
  limits: { fileSize: 50 * 1024 * 1024 },
  createParentPath: true
}));
```

¹ <https://www.npmjs.com/package/express-fileupload>

```
app.set('uploadPath', __dirname)
```

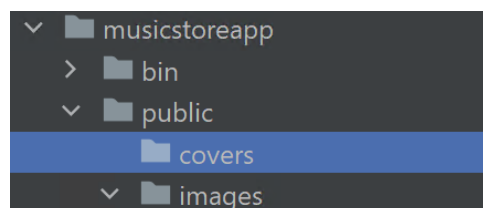
Modificamos la petición **POST /songs**, para que una vez este guardada la canción en la base de datos, no enviar directamente la respuesta. La respuesta se enviará una vez que se complete la subida de los ficheros.

Comprobamos si, en la request, ha llegado un fichero con el nombre portada (**req.files.cover**). 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 en disco. Dependiendo del contenido de la variable **err**, devolveremos una respuesta u otra. En el router **songs** modificamos el método **songsRepository.insertSong()**.

```
songsRepository.insertSong(song, function (songId) {  
  if (songId == null) {  
    res.send("Error al insertar canción");  
  } else {  
    //res.send("Agregada id: " + id);  
    if (req.files != null) {  
      let imagen = req.files.cover;  
      imagen.mv(app.get("uploadPath") + '/public/covers/' + songId + '.png', function (err) {  
        if (err) {  
          res.send("Error al subir la portada de la canción")  
        } else {  
          res.send("Agregada la canción ID: " + songId)  
        }  
      })  
    } else {  
      res.send("Agregada la canción ID: " + songId)  
    }  
  }  
});
```

Asegúrate de borrar la línea **res.send()** que existía de pasos anteriores, porque **no se puede seguir procesando la petición una vez se ha enviado la respuesta.**

Hemos indicado que el fichero de portada se guarda en la carpeta **public/covers**. Esta carpeta se creará automáticamente si no existe, porque hemos configurado el módulo con el parámetro **createParentPath = true**.



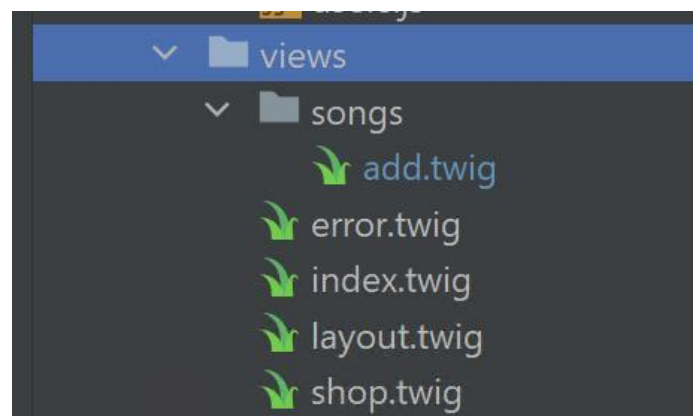


Antes de probarlo debemos asegurarnos de que el formulario contiene el atributo **enctype="multipart/form-data"** con el fin de que pueda procesar ficheros. Abrimos la vista correspondiente **/views/add.twig** y añadimos el atributo y su valor en la declaración, así como el input **file name="cover"**

```
<h2>Agregar canción</h2>
<form class="form-horizontal" method="post" action="/songs/add" enctype="multipart/form-data">
....
<input type="file" id="cover" class="custom-file-input" name="cover"/>
...
```

Estructura de carpeta para agrupar las vistas por entidades

Para ir creando una arquitectura más modular, vamos a ir agrupando los módulos y ficheros por entidades. Empezamos creando una **carpeta songs dentro de views** y movemos el fichero **add.twig** a la carpeta. La estructura de views quedaría como en la siguiente imagen:



Cambiamos el método get **"/song/add"** para que ahora renderice a la nueva ruta donde se encuentra la plantilla:

```
app.get('/songs/add', function (req, res) {
  res.render("songs/add.twig");
});
```

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/covers.**



uoMusic

Agregar canción

Título:
Disciplina

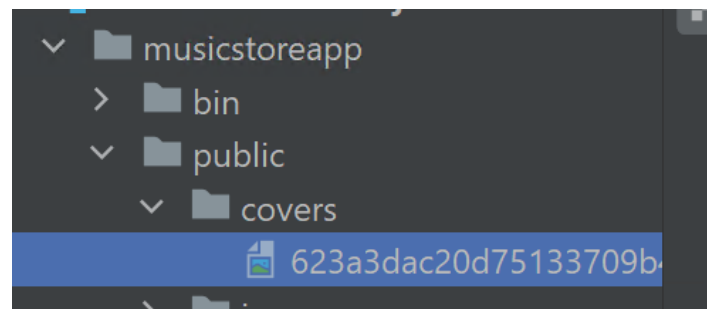
Genero:
Pop

Precio (€):
1

Imagen portada:
Seleccionar archivo song1.png

Fichero audio:
Seleccionar archivo Ninguno ...hivo selec.

Agregar



Nota: Es posible que sea necesario refrescar la caché del navegador para que se muestre el formulario HTML con los cambios añadidos (control + F5).

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 != null) {  
  let imagen = req.files.cover;  
  imagen.mv(app.get("uploadPath") + '/public/covers/' + songId + '.png', function (err) {  
    if (err) {  
      res.send("Error al subir la portada de la canción")  
    } else {  
      //res.send("Agregada la canción ID: " + songId)  
      if (req.files.audio != null) {  
        let audio = req.files.audio;  
        audio.mv(app.get("uploadPath") + '/public/audios/' + songId + '.mp3', function (err) {  
          if (err) {  
            res.send("Error al subir el audio");  
          } else {  
            res.send("Agregada la canción ID: " + songId);  
          }  
        })  
      }  
    }  
  })  
}
```



```
}  
    }  
}  
}  
} else {  
    res.send("Agregada la canción ID: " + songId)  
}
```

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.

Como último paso, modificamos el formulario de la vista `/songs/add.twig` para incluir que los ficheros sean obligatorios (**required**) y acotar el tipo de formatos aceptados. El atributo **accept**² toma como valor una lista separada por comas de uno o más tipos de archivos, que describen qué tipos de archivo permitir.

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

Guardamos los cambios, ejecutamos la aplicación y probamos a añadir una canción. Dentro del fichero **canciones.rar**, que está disponible en la carpeta OneDrive compartida, 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 /shop**.

Actualizar el repositorio

Abrimos el fichero **songsRepository.js** y agregamos una nueva función **getSongs()**.

² <https://developer.mozilla.org/es/docs/Web/HTML/Attributes/accept>



La función **getSongs()** va a devolver todos los documentos almacenados en la **colección** “songs” a través de la función **find()**. Si no establecemos un filtro en el método **find()**, nos devuelve todas las canciones. El resultado de la consulta lo transformaremos en un array. En este caso vamos a definir la función asíncrona y la ejecutamos utilizando el enfoque **async/await**.

```
module.exports = {
  mongoClient: null,
  app: null,
  init: function (app, mongoClient) {
    this.mongoClient = mongoClient;
    this.app = app;
  },
  getSongs: async function () {
    try {
      const client = await this.mongoClient.connect(this.app.get('connectionStrings'));
      const database = client.db("musicStore");
      const collectionName = 'songs';
      const songsCollection = database.collection(collectionName);
      const songs = await songsCollection.find().toArray();
      return songs;
    } catch (error) {
      throw (error);
    }
  },
  insertSong: function (song, callbackFunction) {...}
```

Actualizar el controlador

Agregamos la función **GET /shop** en el fichero **routes/songs.js** y realizamos una llamada a **getSongs()**. Una vez recibida la lista de la base de datos, la enviamos como parámetro a la vista **shop.twig** usando la clave **songs**.

```
app.get('/shop', function (req, res) {
  songsRepository.getSongs().then(songs => {
    res.render("shop.twig", {songs: songs});
  }).catch(error => {
    res.send("Se ha producido un error al listar las canciones " + error)
  });
});
```

Si guardamos los cambios y ejecutamos la aplicación podremos ver en <http://localhost:8081/shop> el catálogo de canciones. Siempre y cuando previamente las hayamos guardado desde nuestra aplicación mediante el formulario para añadir canciones.

Actualizar la vista

La vista todavía no muestra las portadas de las canciones. Debemos acceder a **views/shop.twig** y modificar el código HTML correspondiente a la imagen de la portada.

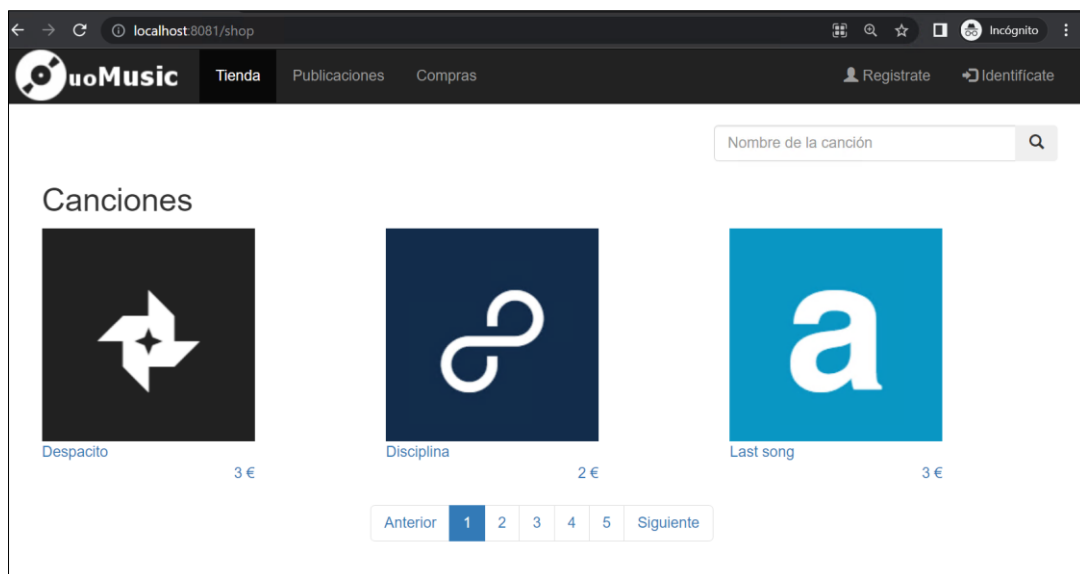


La imagen es un fichero png contenido en la carpeta `/covers/{id de la canción}.png`

El código a añadir en `shop.twig` sería el siguiente:

```
{% for song in songs %}
<div class="col-xs-6 col-sm-6 col-md-4 col-lg-3">
  <div style="width:200px">
    <a href="/songs/id">
      
      <!-- http://www.socicon.com/generator.php -->
    </a>
    <div>{{ song.title }}</div>
    <div class="small">{{ song.author }}</div>
    <div class="text-right">{{ song.price }} €</div>
  </div>
</div>
{% endfor %}
```

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



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 y otras opciones

Actualizar el repositorio

Vamos a introducir los parámetros `filter` y `options`, en la función `getSongs()` de `songsRepository.js`. El primero será el filtro o criterio que deben de cumplir las canciones para ser devueltas (similar al `where` de SQL). El segundo será el opcional y será un objeto



con algunos parámetros como **sort** para ordenar la lista o **projection** para incluir solo los campos especificado en los documentos devueltos. Estos serán utilizados como parámetro en la función **find(<filter>, <options>)**. En el caso de no necesitar un filtro o parámetros opcionales, enviaríamos los objetos vacíos {}.

Por ejemplo, si quisiéramos **obtener el precio de la canción** con título “Black House” el filtro sería:

```
filter = {"title": "Black House"}
```

```
options = { sort: { title: 1 }, projection: { price : 1 } }
```

Nota: Fíjate como en **projection** seleccionamos únicamente el atributo (el price) que queremos del documento, descartando todos los demás (género, título, etcétera).

Los valores 1 y 0 funcionan como true y false y pueden ser intercambiados por éstos.

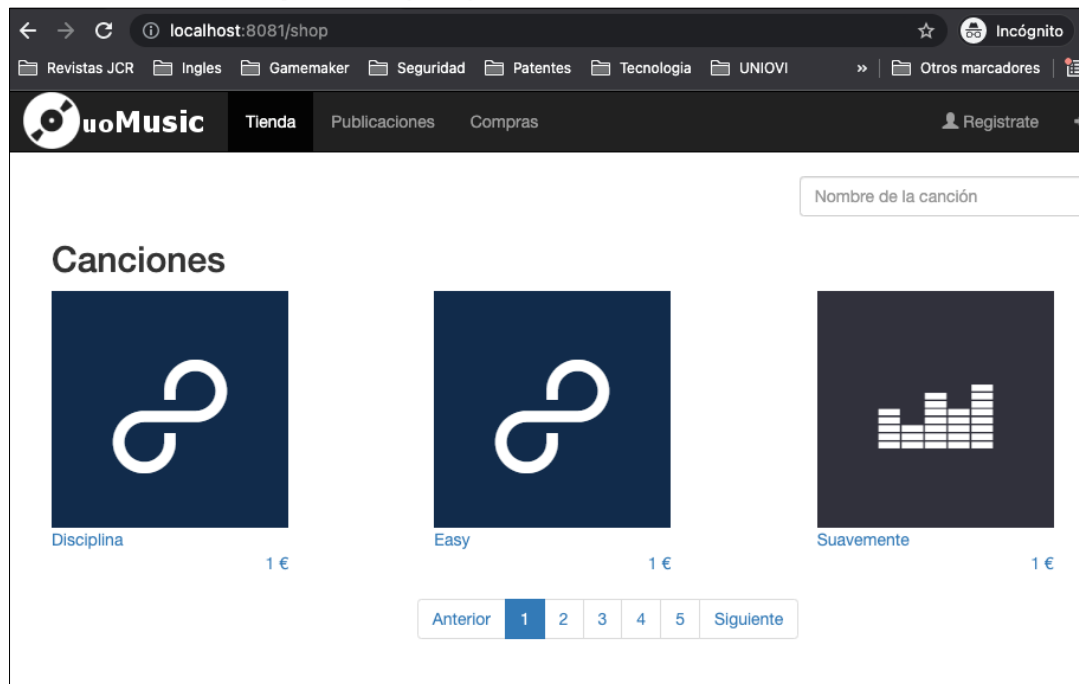
Si, por ejemplo, quisiéramos obtener todos los atributos EXCEPTO el precio, podríamos sustituir price : 1 por price : 0. Recibiríamos el autor, el género, etcétera pero NO el precio.

```
getSongs: async function (filter, options) {  
  try {  
    const client = await this.mongoClient.connect(this.app.get('connectionStrings'));  
    const database = client.db("musicStore");  
    const collectionName = 'songs';  
    const songsCollection = database.collection(collectionName);  
    const songs = await songsCollection.find(filter, options).toArray();  
    return songs;  
  } catch (error) {  
    throw (error);  
  }  
},  
...
```

Actualizar el controlador

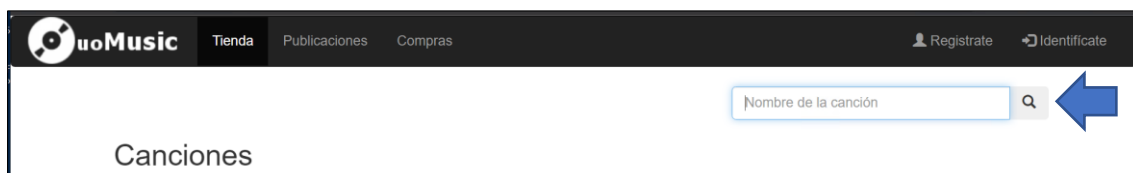
Para listar todas las canciones de la tienda en **GET /shop** el criterio o filtro que aplicamos al llamar a la función **getSong()** será el del objeto vacío y en options le pasamos el objeto indicado que nos devuelva a lista ordenada ascendente, por título (title).

```
...  
app.get('/shop', function (req, res) {  
  let filter = {};  
  let options = {sort: { title: 1}};  
  
  songsRepository.getSongs(filter, options).then(songs => {  
    res.render("shop.twig", {songs: songs});  
  }).catch(error => {  
    res.send("Se ha producido un error al listar las canciones " + error)  
  });  
})
```

Sistema de Búsqueda

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



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

```
app.get('/shop', function (req, res) {  
  let filter = {};  
  let options = {sort: {title: 1}};  
  if(req.query.search != null && typeof(req.query.search) != "undefined" && req.query.search != ""){  
    filter = { "title" : req.query.search };  
  }  
  ....  
})
```

En este caso y con la condición de que hemos aplicado en el **find()**, **el título 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/>

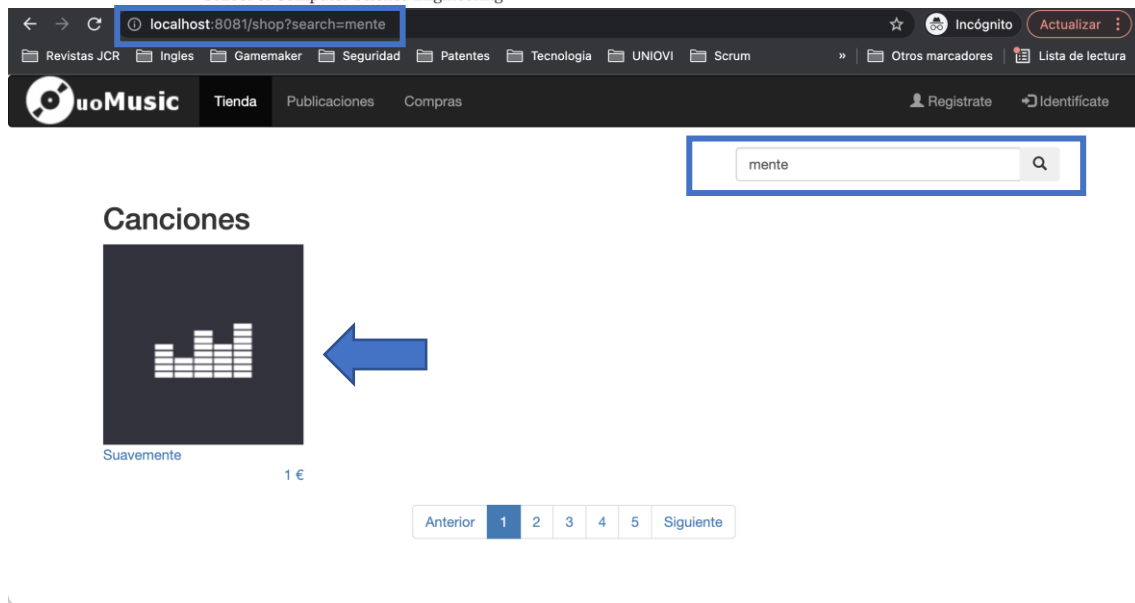
También expresiones regulares/patrones:

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

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

```
app.get('/shop', function (req, res) {
  let filter = {};
  let options = {sort: {title: 1}};
  if (req.query.search != null && typeof (req.query.search) != "undefined" && req.query.search != "") {
    //filter = {"title": req.query.search};
    filter = {"title": {$regex: "." + req.query.search + "."}};
  }
});
```

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



4.3 Vista de detalles de una 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.

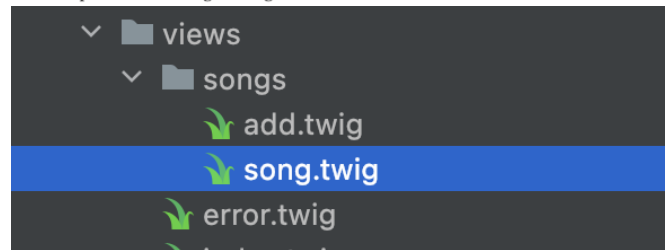
Utilizaremos el `_id` del objeto canción como identificador, por lo que la petición para ver los detalles de una canción será: **GET** `/songs/{id}`. Añadimos el enlace los detalles de la canción en la vista `/views/shop.twig`

```
...
{% for song in songs %}
  <div class="col-xs-6 col-sm-6 col-md-4 col-lg-3">
    <div style="width:200px">
      <a href="/songs/{{song._id}}">
        
        <!-- http://www.socicon.com/generator.php -->
        <div>{{ song.title }} </div>
        <div class="small">{{ song.author }} </div>
        <div class="text-right">{{ song.price }} €</div>
      </a>
    </div>
  </div>
{% endfor %}
...
```

La lógica de la función **GET** `/songs/{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/song.html` que tenemos actualmente en la carpeta `public` del proyecto, a la carpeta `/views/songs` y lo renombramos a `song.twig`.



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

```
{% extends "layout.twig" %}

{% block title %} Detalles de canción {% endblock %}

{% block main_container %}
  <div class="row">
    <div class="media col-xs-10">
      <div class="media-left media-middle">
        
      </div>
      <div class="media-body">
        <h2>{{ song.title }}</h2>
        <p>{{ song.author }}</p>
        <p>{{ song.kind }}</p>
        <button type="button" class="btn btn-primary pull-right">{{ song.price }} €</button>
        <!-- Cambiar el precio por "reproducir" si ya está comprada -->
      </div>
    </div>
  </div>
</div>
```

Actualizar repositorio

A continuación, implementaremos la función **findSong()** en el repositorio **songsRepository.js**. Esta función será la encargada de devolver una única canción en base a un filtro y unas opciones:

```
findSong: async function (filter, options) {
  try {
    const client = await this.mongoClient.connect(this.app.get('connectionStrings'));
    const database = client.db("musicStore");
    const collectionName = 'songs';
    const songsCollection = database.collection(collectionName);
    const song = await songsCollection.findOne(filter, options);
    return song;
  } catch (error) {
    throw (error);
  }
},
...
```



Actualizar controlador

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

```
app.get('/songs/:id', function (req, res) {  
  let filter = { _id: req.params.id};  
  let options = {};  
  songsRepository.findSong(filter, options).then(song => {  
    res.render("songs/song.twig", {song: song});  
  }).catch(error => {  
    res.send("Se ha producido un error al buscar la canción " + error)  
  });  
});
```

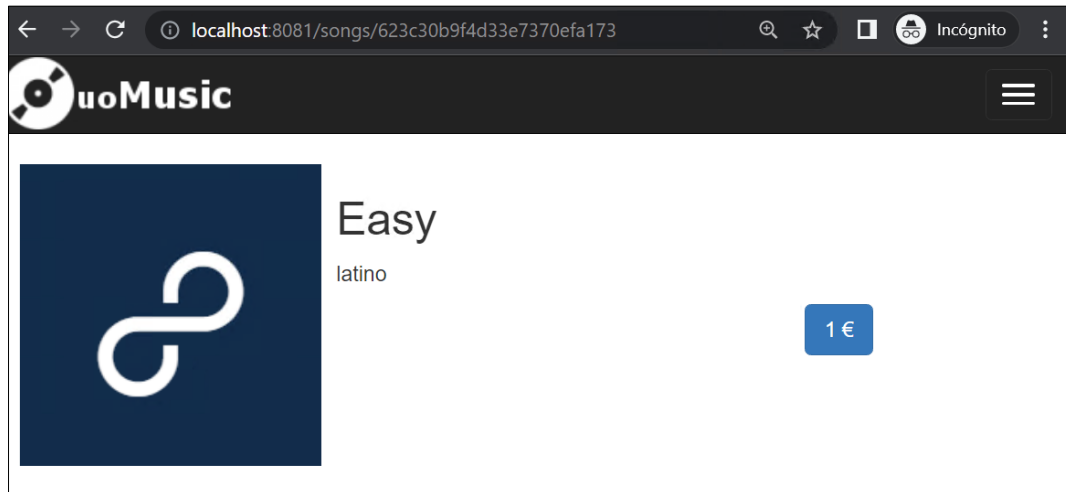
Aunque aparentemente el código esté bien, si guardamos los cambios y ejecutamos la aplicación observamos que **nunca se recupera ninguna canción**. Esto es **debido a que estamos empleando la _id como String cuando debemos tratarla como un objeto (ObjectID)**. Utilizamos la función **ObjectID()** de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

```
app.get('/songs/:id', function (req, res) {  
  // let filter = { _id: req.params.id};  
  let filter = { _id: ObjectID(req.params.id)};  
  let options = {};  
  songsRepository.findSong(filter, options).then(song => {  
    res.render("songs/song.twig", {song: song});  
  }).catch(error => {  
    res.send("Se ha producido un error al buscar la canción " + error)  
  });  
});
```

Para poder utilizar este método tenemos que importar el objeto **ObjectID** del módulo **MongoDB**. Añadimos la siguiente línea al principio del fichero **routes/songs.js**.

```
const {ObjectID} = require("mongodb");  
module.exports = function (app, songsRepository) {
```

Guardamos los cambios, ejecutamos la aplicación y comprobamos que la vista de detalles se muestra correctamente. Desde la URL <http://localhost:8081/shop> hacemos clic sobre el título de una canción y se mostrará el detalle de esta, similar a como se muestra en la siguiente imagen:



Nota: Incluir el siguiente Commit Message ->

“SDI-IDGIT-8.3-Subida a ficheros y gestión de colecciones.”

OJO: sustituir IDGIT por tu número asignado (p.e. 2223-101):

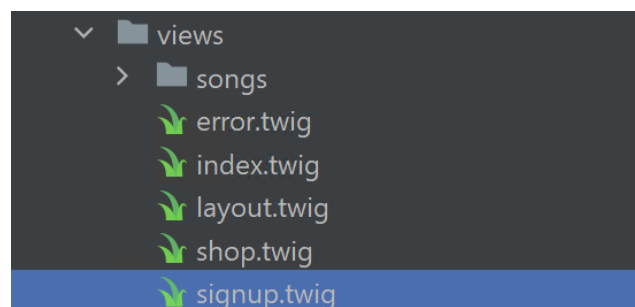
“SDI-2223-101-8.3-Subida a ficheros y gestión de colecciones.”

(No olvides incluir los guiones y NO incluyas BLANCOS al principio o al final, ni las comillas)

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 **“users”**. Almacenaremos los siguientes datos del usuario: **_id (automático), email y password**.

Movemos y renombramos la vista **/public/signup.html** a **/views/signup.twig**



Queremos que, al recibir la petición **GET /users/signup**, se muestre la vista de **signup.twig**. Como se trata de una vista relativa a los usuarios, la modificamos en el controlador **/routes/users.js**. Este módulo fue creado por el generador de Express y es otra forma de crear módulos, usando `express.Router()`. Pero en nuestro caso, vamos a crear



el módulo como una función. Más adelante veremos los routers. Reemplazamos el Código de **routes/users.js** por el siguiente:

```
module.exports = function (app, usersRepository) {  
  app.get('/users', function (req, res) {  
    res.send('lista de usuarios');  
  })  
  app.get('/users/signup', function (req, res) {  
    res.render("signup.twig");  
  })  
}
```

El formulario incluido en **signup.twig** envía una petición **POST /users/signup** con los parámetros **email** y **password**. Por tanto, debemos crear y almacenar un usuario con esos datos. En este formulario, cambiamos el atributo **action** para enviar la petición **POST** a **users/signup**. Podríamos añadir otros datos del usuario, tales como: nombre, apellidos, dirección, etc.

```
{% block main_container %}  
<h2>Registrar usuario</h2>  
<form class="form-horizontal" method="post" action="/users/signup" >  
  <div class="form-group">  
    ....
```

Cifrando el password con Crypto

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

```
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**).

```
app.set('uploadPath', __dirname)  
app.set('clave', 'abcdefg');  
app.set('crypto', crypto);
```

Volvemos a **users.js** e implementamos el endpoint **POST /users/signup**.

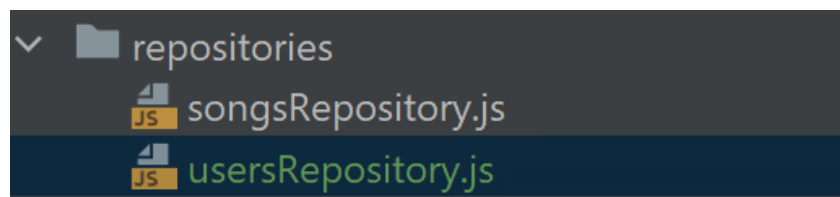
1. Obtenemos el parámetro **req.body.password** y lo ciframos con el módulo **crypto**.
2. Creamos un objeto usuario incluyendo el password seguro.



```
module.exports = function (app) {
  app.get('/users', function (req, res) {
    res.send('lista de usuarios');
  })
  app.get('/users/signup', function (req, res) {
    res.render("signup.twig");
  })
  app.post('/users/signup', function (req, res) {
    let securePassword = app.get("crypto").createHmac('sha256', app.get('clave'))
      .update(req.body.password).digest('hex');
    let user = {
      email: req.body.email,
      password: securePassword
    }
    res.send('usuario registrado');
  });
}
```

Crear repositorio de usuarios

Para poder almacenar los usuarios en base de datos **creamos el módulo userRepository.js** y la función **insertUser()**.



```
module.exports = {
  mongoClient: null,
  app: null,
  init: function (app, mongoClient) {
    this.mongoClient = mongoClient;
    this.app = app;
  }, insertUser: async function (user) {
    try {
      const client = await this.mongoClient.connect(this.app.get('connectionStrings'));
      const database = client.db("musicStore");
      const collectionName = 'users';
      const usersCollection = database.collection(collectionName);
      const result = await usersCollection.insertOne(user);
      return result.insertedId;
    } catch (error) {
      throw (error);
    }
  }
};
```




Actualizar controlador de usuarios

Añadimos la ejecución de la función **insertUser()** en el controlador **routes/users.js** y le pasamos como parámetro el **usersRepository** al módulo.

```
module.exports = function (app, usersRepository) {  
  app.get('/users', function (req, res) {  
    res.send('lista de usuarios');  
  })  
  app.get('/users/signup', function (req, res) {  
    res.render("signup.twig");  
  })  
  app.post('/users/signup', function (req, res) {  
    let securePassword = app.get("crypto").createHmac('sha256', app.get('clave'))  
      .update(req.body.password).digest('hex');  
    let user = {  
      email: req.body.email,  
      password: securePassword  
    }  
    //res.send('usuario registrado');  
    usersRepository.insertUser(user).then(userId => {  
      res.send('Usuario registrado ' + userId);  
    }).catch(error => {  
      res.send("Error al insertar el usuario");  
    });  
  });  
}
```

Incluir repositorio de usuarios en app.js

Ahora incluimos el módulo **usersRepository** en el **app.js** y eliminamos las dos líneas que incluía al anterior modulo **users.js** en **app**.

```
const usersRepository = require("./repositories/usersRepository.js");  
usersRepository.init(app, MongoClient);  
require("./routes/users.js")(app, usersRepository);  
  
let indexRouter = require('./routes/index');  
//let usersRouter = require('./routes/users');  
....  
app.use('/', indexRouter);  
//app.use('/users', usersRouter);
```

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

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



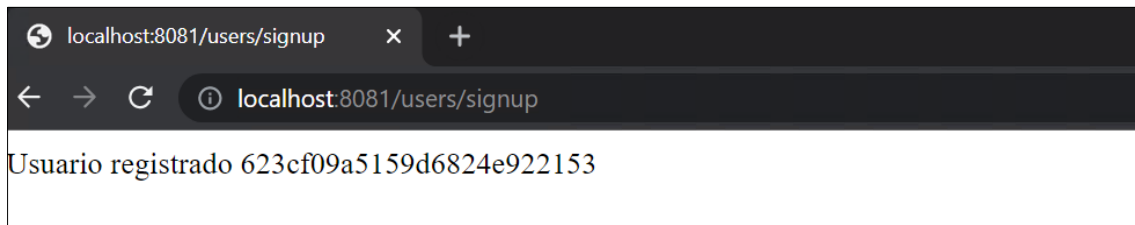
- Usuario prueba2@prueba2.com (password: prueba2).

Registrar usuario

Email:

Password:

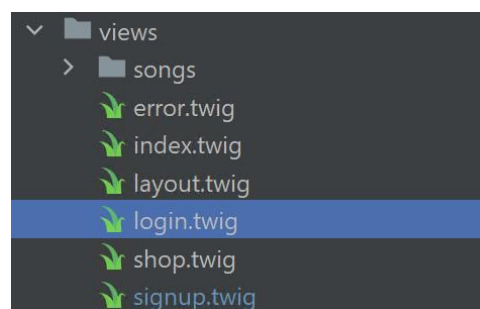
Registrar



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 notificar al usuario que el email ya está en uso. Todavía no lo hemos implementado.

6 Autenticación de usuario

En este apartado, añadiremos el proceso de autenticación de usuarios. Como primer paso, movemos y renombramos la vista `/public/login.html` a la carpeta `views/login.twig`.



En el fichero `views/login.twig` hay definido un formulario **POST** `/users/login` que envía los parámetros **email** y **password**. Modificamos el atributo `action` para que envíe la petición a `users/login`.

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



Actualizar el repositorio de usuarios

El formulario anterior envía una petición **POST /users/login** con los parámetros **email** y **password**. Por tanto, comprobaremos si existe un usuario con ese email y password (encriptado) realizando un **findOne()**. Si devuelve un error o 0 resultados, los datos son erróneos. Vamos a implementar una nueva función **findUser()** en el **usersRepository.js**. El proceso será ser el mismo que para **findSong()**, incluyendo un filtro y unas opciones configurable:

```
module.exports = {
  mongoClient: null,
  app: null,
  init: function (app, mongoClient) {
    this.mongoClient = mongoClient;
    this.app = app;
  }, findUser: async function (filter, options) {
    try {
      const client = await this.mongoClient.connect(this.app.get('connectionStrings'));
      const database = client.db("musicStore");
      const collectionName = 'users';
      const usersCollection = database.collection(collectionName);
      const user = await usersCollection.findOne(filter, options);
      return user;
    } catch (error) {
      throw (error);
    }
  },
  insertUser: async function (user) {..}
```

Actualizar el controlador de usuarios

Accedemos al controlador **users.js** e implementamos la respuesta a **GET /users/login**.

```
app.get('/users/login', function (req, res) {
  res.render("login.twig");
})
```

Para finalizar, el método **POST /users/login** recibirá los datos del usuario (enviados mediante el formulario) y realizará una búsqueda a través de **usersRepository.findUser(filter, options)**. Si la consulta devuelve un usuario, el proceso de autenticación ha sido realizado correctamente.



```
app.post('/users/login', function (req, res) {  
  let securePassword = app.get("crypto").createHmac('sha256', app.get('clave'))  
    .update(req.body.password).digest('hex');  
  let filter = {  
    email: req.body.email,  
    password: securePassword  
  }  
  let options = {};  
  usersRepository.findUser(filter, options).then(user => {  
    if (user == null) {  
      res.send("Usuario no identificado");  
    } else {  
      res.send("Usuario identificado correctamente: " + user.email);  
    }  
  }).catch(error => {  
    res.send("Se ha producido un error al buscar el usuario: " + error)  
  })  
})
```

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

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

← → ↻ ⓘ localhost:8081/users/login 🔍 ☆ □ Incógnito

uoMusic Tienda Publicaciones Compras Registrare Identificate

Identificación de usuario

Email:

Password:

← → ↻ ⓘ localhost:8081/users/login 🔍 ☆ □ Incógnito

Usuario identificado correctamente: prueba2@prueba2.com



7 Uso de sesión

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

```
PS C:\Dev\Projects\SDI_projects\sdiIDGIT-lab-nodejs\musicstoreapp> npm install express-session
added 6 packages, and audited 103 packages in 3s

5 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Dev\Projects\SDI_projects\sdiIDGIT-lab-nodejs\musicstoreapp>
```

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>

```
let app = express();

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

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 /users/login** para que, si el usuario se identifica correctamente, almacenar en sesión su **email** (porque es un identificador único).

```
app.post('/users/login', function (req, res) {
  let securePassword = app.get("crypto").createHmac('sha256', app.get('clave'))
    .update(req.body.password).digest('hex');
  let filter = {
    email: req.body.email,
    password: securePassword
  }
  let options = {};
  usersRepository.findUser(filter, options).then(user => {
    if (user == null) {
      req.session.user = null;
      res.send("Usuario no identificado");
    } else {
      req.session.user = user.email;
      res.send("Usuario identificado correctamente: " + user.email);
    }
  });
});
```



```
}  
}).catch(error => {  
  req.session.user = null;  
  res.send("Se ha producido un error al buscar el usuario: " + error)  
})  
})
```

Desconectarse

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

```
app.get('/users/logout', function (req, res) {  
  req.session.user = null;  
  res.send("El usuario se ha desconectado correctamente");  
})
```

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

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

```
app.post('/songs/add', function (req, res) {  
  if ( req.session.user == null){  
    res.redirect("/shop");  
    return;  
  }  
  let song = {  
    title: req.body.title,  
    kind: req.body.kind,  
    price: req.body.price,  
    author: req.session.user  
  }  
  ...
```

También debemos comprobar si hay un usuario identificado en **GET /songs/add**

```
app.get('/songs/add', function (req, res) {  
  if ( req.session.user == null){  
    res.redirect("/shop");  
    return;  
  }  
  res.render("songs/add.twig")  
});
```



Nos aseguramos que en la vista **shop.twig** se muestra el nombre del autor.

```
{% for song in songs %}
<div class="col-xs-6 col-sm-6 col-md-4 col-lg-3">
  <div style="width:200px">
    <a href="/songs/{{ song._id }}">
      
      <div>{{ song.title }} </div>
      <div class="small">{{ song.author }} </div>
      <div class="text-right">{{ song.price }} €</div>
    </a>
  </div>
</div>
{% endfor %}
```

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/logout>
- Intentamos acceder a <http://localhost:8081/songs/add> sin usuario en sesión y nos debería redirigir a la tienda(/shop) (no hemos iniciado sesión).
- Nos identificamos en <http://localhost:8081/users/login>
- Agregamos una canción <http://localhost:8081/songs/add>
- Buscamos la canción en la tienda <http://localhost:8081/shop> y verificamos que tiene un **autor**





Nota: Incluir el siguiente Commit Message ->

“SDI-IDGIT-8.4-Registro de usuario, autenticación y uso de sesión.”

OJO: sustituir IDGIT por tu número asignado (p.e. 2223-101):

“SDI-2223-101-8.4-Registro de usuario, autenticación y uso de sesión.”

(No olvides incluir los guiones y NO incluyas BLANCOS al principio o al final, ni las comillas)

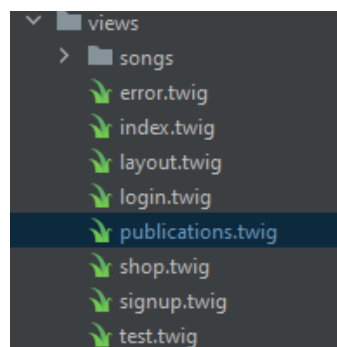
8 Colecciones relacionadas – Usuarios y canciones

En este apartado, vamos a implementar una respuesta (**GET /publications**) que devuelva únicamente las canciones publicadas por el autor (usuario) actualmente identificado en sesión. El código será como el de **GET /shop**, pero con la particularidad de que necesitamos un filtro `{ author : req.session.user }`. Añadimos el siguiente **método GET** en el fichero `/routes/songs.js`:

```
app.get('/publications', function (req, res) {  
  let filter = {author : req.session.user};  
  let options = {sort: {title: 1}};  
  songsRepository.getSongs(filter, options).then(songs => {  
    res.render("shop.twig", {songs: songs});  
  }).catch(error => {  
    res.send("Se ha producido un error al listar las publicaciones del usuario:" + error)  
  });  
});
```

Si nos identificamos con un usuario y accedemos a <http://localhost:8081/publications> veremos que únicamente aparecen las canciones en las que figura como autor.

En lugar de reutilizar la vista **shop.twig** vamos a usar **publications.twig** Esta vista muestra información de las canciones, pero en otro formato. Movemos y renombramos la vista de la carpeta `/public/publications.html` a la carpeta `/views/publications.twig`





Actualizar controlador

Cambiamos el nombre de la vista en el método **GET /publications**.

```
app.get('/publications', function (req, res) {  
  let filter = {author: req.session.user};  
  let options = {sort: {title: 1}};  
  songsRepository.getSongs(filter, options).then(songs => {  
    res.render("publications.twig", {songs: songs});  
  }).catch(error => {  
    res.send("Se ha producido un error al listar las publicaciones del usuario: " + error)  
  });  
})
```

Actualizar vista publicaciones

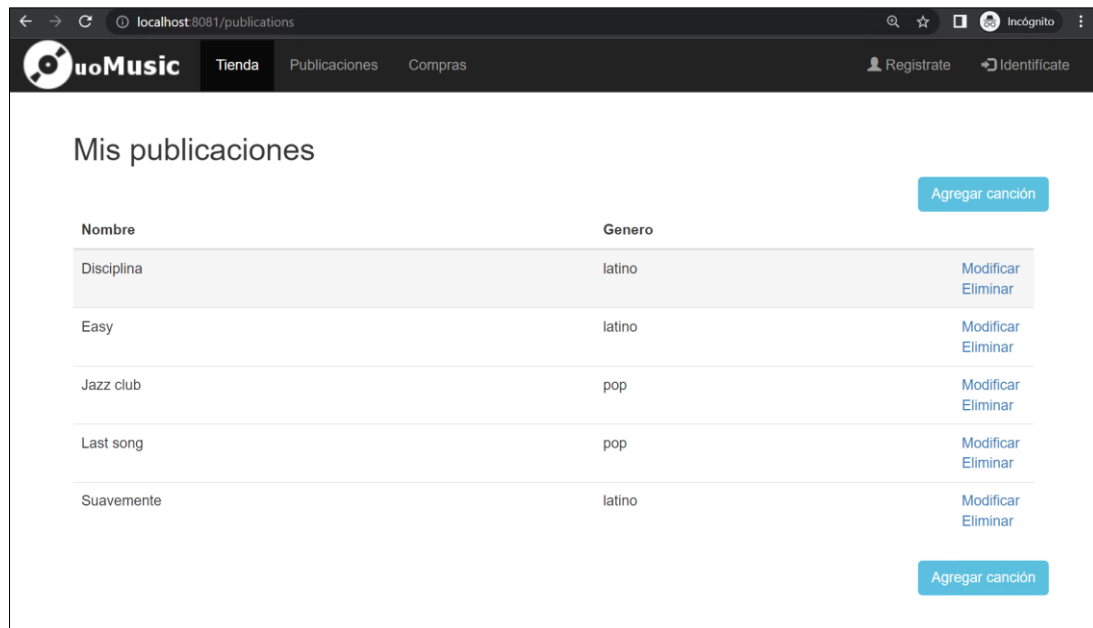
Actualizamos el fichero publications.twig, copiando el siguiente contenido:

```
{% extends "layout.twig" %}  
{% block title %} Mis publicaciones {% endblock %}  
{% block main_container %}  
  <h2>Mis publicaciones</h2>  
  
  <!-- Agregar Canción -->  
  <div class="row text-right">  
    <a href="/songs/add" class="btn btn-info" role="button">Agregar canción</a>  
  </div>  
  
  <div class="table-responsive">  
    <table class="table table-hover">  
      <thead>  
        <tr>  
          <th>Nombre</th>  
          <th>Genero</th>  
          <th class="col-md-1"></th>  
        </tr>  
      </thead>  
      <tbody>  
        {% for song in songs %}  
          <tr>  
            <td>{{ song.title }}</td>  
            <td>{{ song.kind }}</td>  
            <td><a href="/songs/edit/{{ song._id }}">Modificar</a> <br>  
              <a href="/songs/delete/{{ song._id }}">Eliminar</a> <br>  
            </td>  
          </tr>  
        {% endfor %}  
      </tbody>  
    </table>  
  </div>
```



```
<div class="row text-right">
  <a href="/songs/add" class="btn btn-info" role="button">Agregar canción</a>
</div>
{% endblock %}
```

Si accedemos a <http://localhost:8081/publications>, 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 /songs/add** y **POST /songs/add**.

```
app.get('/songs/add', function (req, res) {
  if (req.session.user == null) {
    res.redirect("/shop");
    return;
  }
  res.render("songs/add.twig")
});
```

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 resolver esto mediante la implementación de un enrutador (router).

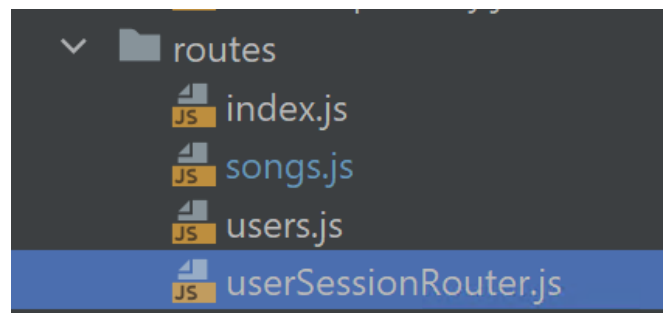


Definimos el router *SessionRouter*

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 /users/login**

Primero creamos el fichero **/routes/userSessionRouter.js** y añadimos el siguiente código:



```
const express = require('express');
const userSessionRouter = express.Router();
userSessionRouter.use(function(req, res, next) {
  console.log("routerUsuarioSession");
  if ( req.session.user ) {
    // dejamos correr la petición
    next();
  } else {
    console.log("va a: " + req.originalUrl);
    res.redirect("/users/login");
  }
});
module.exports = userSessionRouter;
```

Una vez definido el router, debemos especificar sobre qué URLs se aplica el interceptor, mediante **app.use(ruta, router)**. Al especificar rutas, debemos tener en cuenta que captura cualquier tipo de petición (GET, POST, etcétera). En este caso lo aplicaremos sobre **/audios/** (ficheros contenidos en public/audios), **/publications** y **/songs/add**.

Incluimos el router 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 /users y /songs**, ya que si lo declaramos después estaríamos indicando que la gestión de los controladores tiene prioridad.

```
...
app.set('connectionStrings', url);

const userSessionRouter = require('./routes/userSessionRouter');

app.use("/songs/add", userSessionRouter);
app.use("/publications", userSessionRouter);
app.use("/audios/", userSessionRouter);
app.use("/shop/", userSessionRouter)

const songsRepository = require("./repositories/songsRepository.js");
songsRepository.init(app, MongoClient);
require("./routes/songs.js")(app, songsRepository);
```

Ahora eliminamos las condiciones que comprobaban si había un usuario en sesión en GET /songs/add y POST /songs/add en el módulos routes/songs, ya que pasa a ser un código redundante.

```
app.get('/songs/add', function (req, res) {
  if (req.session.user == null) {
    res.redirect("/shop");
    return;
  }
  res.render("songs/add.twig")
});
```

```
app.post('/songs/add', function (req, res) {
  if (req.session.user == null) {
    res.redirect("/shop");
    return;
  }

  let song = {
  ...
}
```

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

- <http://localhost:8081/songs/add>
- <http://localhost:8081/publications>
- <http://localhost:8081/audios> o a un fichero contenido en /audios



El resultado esperado es que nos redirija a /identificarse, tal y como se muestra en la siguiente imagen:

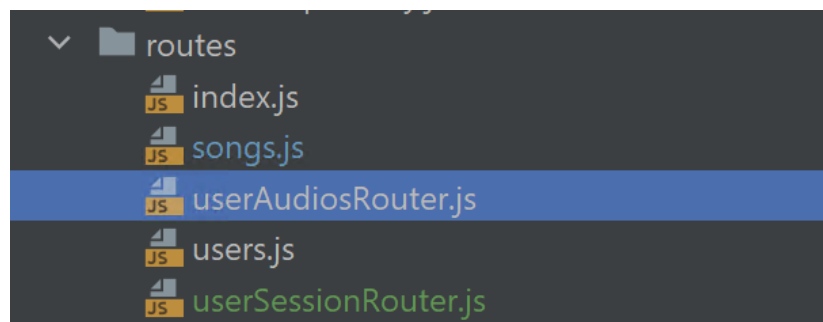
Nota: No obstante, sigue habiendo un problema que pasa inadvertido en algunos sitios web. **En los recursos /audio/ comprobamos que el usuario está en sesión, pero no que ese usuario tiene permiso para acceder al fichero de la canción.** Por ejemplo, solo debería tener acceso si es el autor o si la ha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

Implementar el módulo *userAudiosRouter*

Eliminamos la ruta `/audio/` de los “use” de `userSessionRouter` y creamos un nuevo **router** `userAudiosRouter`. Este nuevo router, obtendrá de la URL el nombre del fichero mp3 (que coincide con la ID de la canción). Por ejemplo:

- `http://localhost:8081/audios/59949ef1152dacadca2f6.mp3`

El módulo de Node.js **path** (<https://Node.js.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.user**). Dejamos avanzar la petición si el usuario coincide, en caso contrario lo enviamos a la tienda `/shop`.



Al fichero le añadimos el siguiente código:

```
const express = require('express');
const {ObjectId} = require('mongodb');
const songsRepository = require("../repositories/songsRepository");
```



```
const userAudiosRouter = express.Router();

userAudiosRouter.use(function (req, res, next) {
  console.log("routerAudios");
  let path = require('path');
  let songId = path.basename(req.originalUrl, '.mp3');
  let filter = { _id: ObjectId(songId)};
  songsRepository.findSong(filter, {}).then(song => {
    if (req.session.user && song.author == req.session.user) {
      next();
    } else {
      res.redirect("/shop");
    }
  }).catch(error => {
    res.redirect("/shop");
  });
});

module.exports = userAudiosRouter
```

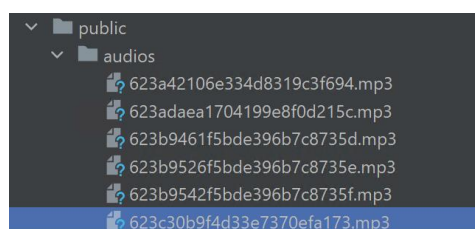
Incluir userAudiosRouter en app.js

Incluimos el router en app.js y cambiamos el app.use de audio para que ahora utilice el nuevo router:

```
const userSessionRouter = require('./routes/userSessionRouter');
const userAudiosRouter = require('./routes/userAudiosRouter');
app.use("/songs/add", userSessionRouter);
app.use("/publications", userSessionRouter);
//app.use("/audios/", userSessionRouter);
app.use("/audios/", userAudiosRouter);
app.use("/shop/", userSessionRouter)
```

Nota: En este punto, hay que **tener cuidado con las canciones que no tengan autor, creadas en apartados anteriores**. El código song.author podría lanzar una excepción, por lo que es **aconsejable que limpien la BD y la rellenen con un par de canciones especificando un autor**.

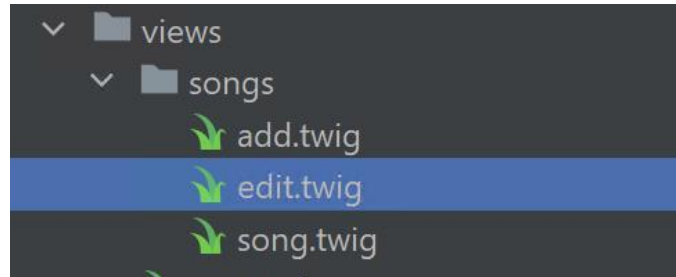
Guardamos los cambios, ejecutamos y comprobamos el funcionamiento. Estando identificados, intentamos acceder a un audio del que no somos propietarios. Podemos ver las ids desde los ficheros del proyecto, por ejemplo, <http://localhost:8081/audios/623c30b9f4d33e7370efa173.mp3>





10 Editar canciones

En este último apartado, vamos a incluir la funcionalidad necesaria para poder modificar las canciones creadas. Como primer paso, renombramos y renombramos la vista **update.html** de la carpeta **/public** a la carpeta **/views/songs/edit.twig**.



songs/edit.twig es una combinación de las vistas utilizadas en la petición **/songs/add**. El cambio fundamental es que el formulario aparecerá con los valores de la canción a modificar y lo envía a **POST /songs/edit/:id**. A continuación, actualizamos el fichero **songs/edit.twig**:

```
{% extends "layout.twig" %}

{% block title %} Agregar canción {% endblock %}

{% block main_container %}
<h2>Modificar canción</h2>
<form class="form-horizontal" method="post" action="/songs/edit/{{ song._id }}"
encType="multipart/form-data">
  <div class="form-group">
    <label class="control-label col-sm-2" for="title">Titulo:</label>
    <div class="col-sm-10">
      <input id="title" type="text" class="form-control" name="title"
placeholder="Nombre de mi canción" required="true"
value="{{ song.title }}" />
    </div>
  </div>
  <div class="form-group">
    <label class="control-label col-sm-2" for="kind">Genero:</label>
    <div class="col-sm-10">
      <select id="kind" class="form-control" name="kind" required="true">
        <option value="pop" id="pop">Pop</option>
        <option value="folk" id="folk">Folk</option>
        <option value="rock" id="rock">Rock</option>
        <option value="reagge" id="reagge">Reagge</option>
        <option value="rap" id="rap">Hip-hop Rap</option>
        <option value="latino" id="latino">Latino</option>
        <option value="blues" id="blues">Blues</option>
        <option value="otros" id="otros">Otros</option>
        <script> document.getElementById("{{ song.kind }}").selected = "true"; </script>
      </select>
    </div>
  </div>
</div>
<div class="form-group">
```



```
<label class="control-label col-sm-2" for="price">Precio (€):</label>
<div class="col-sm-10">
  <input id="price" type="number" step="0.01" class="form-control" name="price"
    placeholder="2.50" required value="{{ song.price }}" />
</div>
</div>
<div class="form-group">
  <label class="control-label col-sm-2" for="cover">Imagen portada:</label>
  <div class="col-sm-10">
    
    <input id="cover" type="file" class="custom-file-input" name="cover"
      accept=".png" />
  </div>
</div>
<div class="form-group">
  <label class="control-label col-sm-2" for="audio">Fichero audio:</label>
  <div class="col-sm-10">
    <audio controls>
      <source src="/audios/{{ song._id }}.mp3" type="audio/mpeg">
    </audio>
    <input id="audio" type="file" class="custom-file-input" name="audio"
      accept=".mp3" />
  </div>
</div>
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-10">
    <button type="submit" class="btn btn-primary">Modificar</button>
  </div>
</div>
</form>
{% endblock %}
```

Actualizar el repositorio de canciones

Antes de implementar las respuestas **GET** y **POST /songs/edit/:id**, nos dirigimos al **songsRepository.js** e implementamos la función **updateSong()**. Para actualizar un documento de una colección utilizamos **collection.updateOne()**, 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 que 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!**

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




```
this.mongoClient = MongoClient;
this.app = app;
},
updateSong: async function(newSong, filter, options) {
  try {
    const client = await this.mongoClient.connect(this.app.get('connectionStrings'));
    const database = client.db("musicStore");
    const collectionName = 'songs';
    const songsCollection = database.collection(collectionName);
    const result = await songsCollection.updateOne(filter, {$set: newSong}, options);
    return result;
  } catch (error) {
    throw (error);
  }
}, ...
```

Actualizar el controlador de canciones

Implementamos en el controlador la respuesta a **GET** /songs/edit/:id en el fichero **routes/songs.js**, donde obtenemos la canción con la id que se recibe como parámetro y se carga la vista **songs/edit.twig**

```
app.get('/songs/edit/:id', function (req, res) {
  let filter = { _id: ObjectId(req.params.id)};
  songsRepository.findSong(filter, {}).then(song => {
    res.render("songs/edit.twig", {song: song});
  }).catch(error => {
    res.send("Se ha producido un error al recuperar la canción " + error)
  });
});
```

A continuación, implementamos la respuesta a **POST** /songs/edit/:id. Obtenemos la **_id** a partir del parámetro: id y creamos un objeto canción con los nuevos valores.

```
app.post('/songs/edit/:id', function (req, res) {
  let song = {
    title: req.body.title,
    kind: req.body.kind,
    price: req.body.price,
    author: req.session.user
  };
  let songId = req.params.id;
  let filter = { _id: ObjectId(songId)};
  //que no se cree un documento nuevo, si no existe
  const options = {upsert: false};
  songsRepository.updateSong(song, filter, options).then(result => {
    res.send("Se ha modificado " + result.modifiedCount + " registro");
  });
});
```



La función está casi completa, pero falta comprobar si la petición contiene los ficheros **req.files.cover** 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. Si 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 /songs/edit/:id**, el código quedará más claro y fácil de modificar si sacamos los pasos a otras funciones.

```
app.post('/songs/edit/:id', function (req, res) {
  let song = {
    title: req.body.title,
    kind: req.body.kind,
    price: req.body.price,
    author: req.session.user
  }
  let songId = req.params.id;
  let filter = { _id: ObjectId(songId)};
  //que no se cree un documento nuevo, si no existe
  const options = {upsert: false}
  songsRepository.updateSong(song, filter, options).then(result => {
    //res.send("Se ha modificado " + result.modifiedCount + " registro");
    step1UpdateCover(req.files, songId, function (result) {
      if (result == null) {
        res.send("Error al actualizar la portada o el audio de la canción");
      } else {
        res.send("Se ha modificado el registro correctamente");
      }
    });
  });
}).catch(error => {
  res.send("Se ha producido un error al modificar la canción " + error);
});
})

function step1UpdateCover(files, songId, callback) {
  if (files && files.cover != null) {
    let image = files.cover;
    image.mv(app.get("uploadPath") + '/public/covers/' + songId + '.png', function (err) {
      if (err) {
        callback(null); // ERROR
      }
    });
  }
}
```

```
    } else {  
      step2UpdateAudio(files, songId, callback); // SIGUIENTE  
    }  
  });  
} else {  
  step2UpdateAudio(files, songId, callback); // SIGUIENTE  
}  
};  
  
function step2UpdateAudio(files, songId, callback) {  
  if (files && files.audio != null) {  
    let audio = files.audio;  
    audio.mv(app.get("uploadPath") + '/public/audios/' + songId + '.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/publications>, podemos modificar una de nuestras canciones.

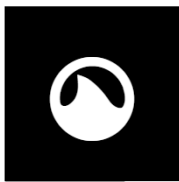
Modificar canción

Nombre:

Genero:

Precio (€):

Imagen portada:


Seleccionar archivo Ningún archivo seleccionado

Fichero audio:

▶ 0:00 / 1:45 🔊 🔍 ⬇

Seleccionar archivo Ningún archivo seleccionado

← → ↻ ⓘ localhost:8081/songs/edit/623d96c63224d74bd9af0075 🔍 ☆

Se ha modificado el registro correctamente



Nota: Incluir el siguiente Commit Message ->

“SDI-IDGIT-8.5-Colecciones relacionadas y routers.”

OJO: sustituir IDGIT por tu número asignado (p.e. 2223-101):

“SDI-2223-101-8.5-Colecciones relacionadas y routers.”

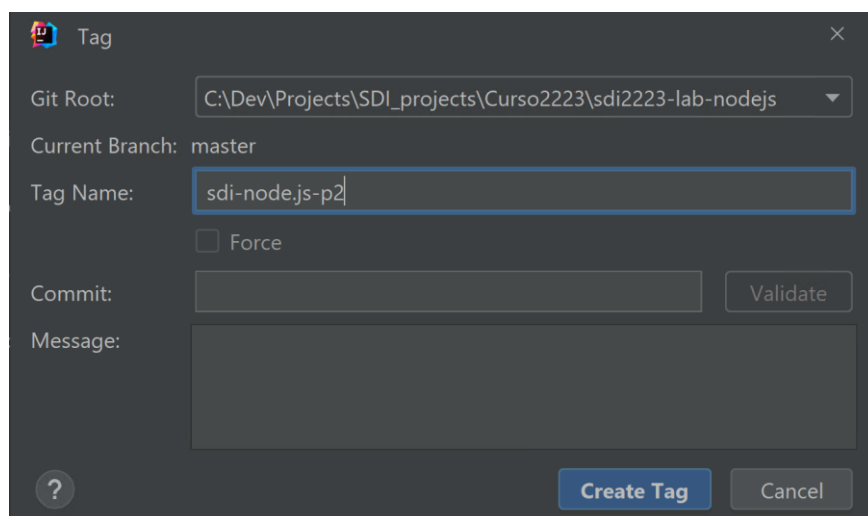
(No olvides incluir los guiones y NO incluyas BLANCOS al principio o al final, ni las comillas)

11 Etiquetar el proyecto

Nota: Etiquetar el proyecto en este punto con la siguiente etiqueta → **sdi-node.js-p2**.

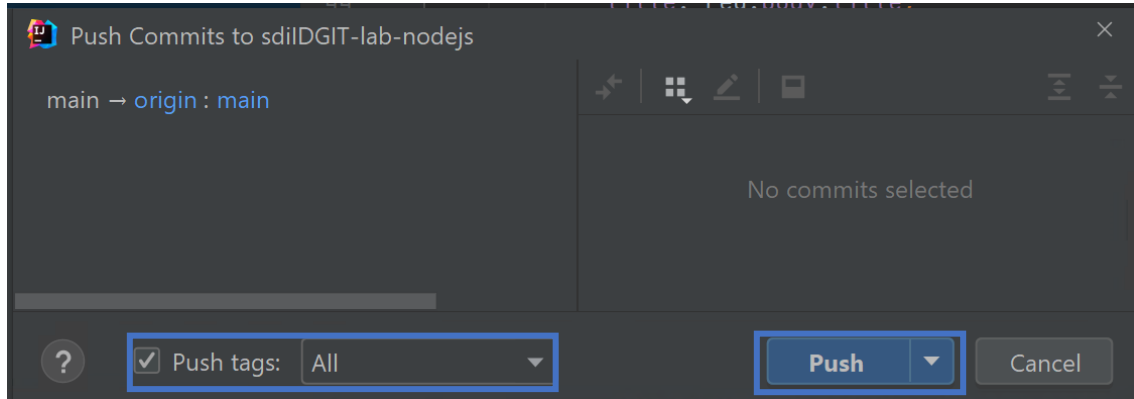
Para crear una **Release** en un proyecto que está almacenado en un repositorio de control de versiones lo común es utilizar etiquetas (Tags), que nos permitirá marcar el avance de un proyecto en un punto dado (una funcionalidad, una versión del proyecto, etc). Para crear una etiqueta en el proyecto en GitHub lo primero que vamos a hacer es hacer clic derecho sobre el proyecto en WebStorm e ir a la opción **Git → New Tag...**

En la siguiente ventana indicamos el nombre de la etiqueta(**sdi-node.js-p2**), un mensaje que indique la funcionalidad que se está etiquetado en este punto. Además, **añadimos el commit actual para crear la etiqueta escribiendo HEAD**. Pulsamos el botón **“Create Tag”**.

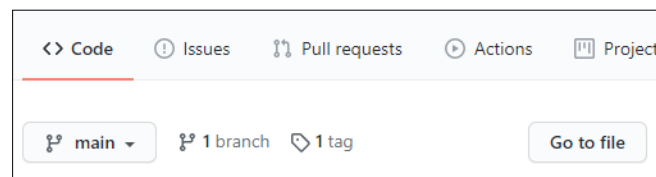




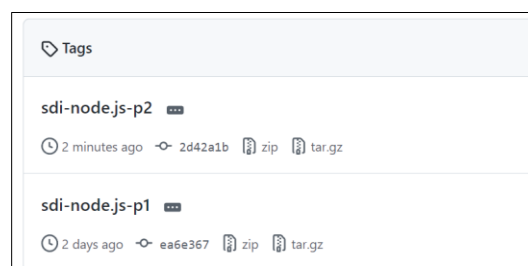
A continuación, vamos a hacer un push a la etiqueta: **Clic derecho en el proyecto → Git → Push...** En el cuadro que aparece, **seleccionamos Push tags: All** y hacemos clic en el botón **Push**.



Una vez subida las etiquetas al repositorio, podemos verificar dichas etiquetas en nuestra cuenta y repositorio de GitHub. **Vamos al repositorio correspondiente y pulsamos en el enlace de tags.**



Si se ha realizado la operación correctamente, debería ser visible la etiqueta creada anteriormente:



12 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.

- ⇒ SDI-2223-101-8.1-Acceso a datos con MongoDB desde Node.js.
- ⇒ SDI-2223-101-8.2-Arquitectura para el acceso a datos.
- ⇒ SDI-2223-101-8.3-Subida a ficheros y gestión de colecciones.
- ⇒ SDI-2223-101-8.4-Registro de usuario, autenticación y uso de sesión
- ⇒ SDI-2223-101-8.5-Colecciones relacionadas y routers