



Sistemas Distribuidos e Internet

Desarrollo de aplicaciones Web con NodeJS

Sesión - 9

Curso 2019/ 2020



Contenido

1	Introducción.....	3
1.1	Instalación y ejecución de nodemon.....	3
2	Eliminar canciones	4
3	Redirección de peticiones.....	6
4	Mensajes y alertas	7
5	Sistema de compra	9
6	Sistema de paginación	13
7	Manejo de errores en la aplicación	16
8	Implementando el protocolo HTTPS.....	17
9	Resultado esperando el repositorio de GitHub.....	18



1 Introducción

En esta sesión de prácticas finalizaremos la implementación restante relativa a la gestión de canciones. Además, implementaremos un sistema de paginación y haremos la aplicación segura mediante el uso de HTTPS.

1.1 Instalación y ejecución de nodemon

nodemon¹ es una herramienta que reinicia automáticamente el servidor de la aplicación NodeJS cuando se detectan cambios en los archivos. **Instalaremos nodemon de manera global** en el sistema usando npm. Ejecutaremos en la consola el siguiente comando:

```
npm install -g nodemon
```

Para arrancar la aplicación con nodemon vamos al directorio del proyecto y ejecutamos el siguiente comando:

```
nodemon app.js
```

```
C:\Dev\workspaces\NodeApps\sdi-lab-node>nodemon app.js
[nodemon] 1.18.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
Servidor activo
```

A partir de ahora, **cada vez que modifiquemos y guardemos un fichero de la aplicación, el servidor se reiniciará automáticamente**. Es posible reiniciarla manualmente mediante el comando "rs". Cuando nodemon detecte un cambio en un fichero mostrará este mensaje:

```
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
Servidor activo
```

Nota: Si arrancamos la aplicación mediante nodemon, no podremos arrancarla a su vez con el IDE, puesto que estaremos ocupando el mismo puerto. Por tanto, **si queremos arrancar la aplicación el IDE para depurar, tendremos que detener primero el proceso lanzado con nodemon**.

```
Debugger listening on ws://127.0.0.1:50001/c6e97979-1368-4d46-8801-ad94244fc43a
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
events.js:174
    throw er; // Unhandled 'error' event
    ^

Error: listen EADDRINUSE: address already in use :::8081
    at Server.setupListenHandle [as _listen2] (net.js:1277:14)
    at listenInCluster (net.js:1325:12)
```

¹ <https://www.npmjs.com/package/nodemon>



2 Eliminar canciones

En este apartado, implementaremos el proceso para eliminar canciones de nuestra aplicación. Para ello, añadiremos un método al controlador que procese las peticiones y otro método que permita eliminar canciones en la base de datos.

Como primer paso, añadimos al controlador **rcanciones.js** una función que procese peticiones **GET /cancion/eliminar/:id**. El parámetro **id** pertenecerá a la canción que queremos eliminar y se lo reenviaremos a **gestorBD.eliminarCancion()**. El código a añadir al controlador es:

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

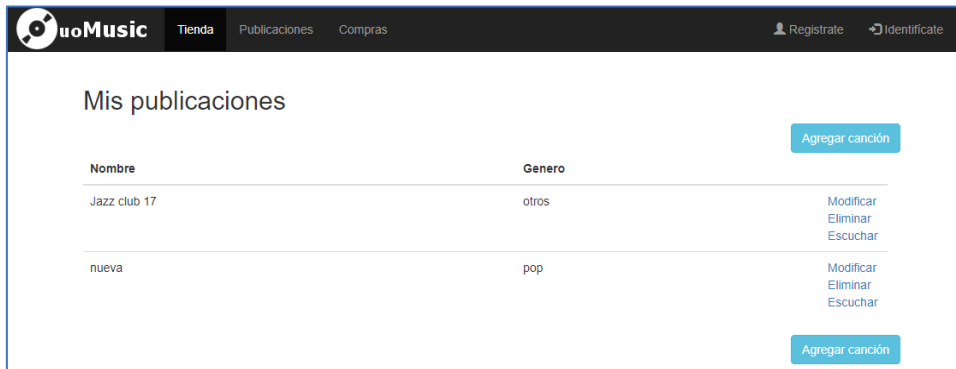
  gestorBD.eliminarCancion(criterio, function(canciones){
    if ( canciones == null ){
      res.send(respuesta);
    } else {
      res.redirect("/publicaciones");
    }
  });
});
```

Implementamos la función **eliminarCanciones** en el **gestorBD.js**:

```
module.exports = {
  mongo : null,
  app : null,
  init : function(app, mongo) {
    this.mongo = mongo;
    this.app = app;
  },
  eliminarCancion : function(criterio, funcionCallback) {
    this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {
      if (err) {
        funcionCallback(null);
      } else {
        let collection = db.collection('canciones');
        collection.remove(criterio, function(err, result) {
          if (err) {
            funcionCallback(null);
          } else {
            funcionCallback(result);
          }
        });
        db.close();
      }
    });
  }
};
```



Sí ejecutamos la aplicación podemos observar que las canciones se eliminan correctamente.



Sin embargo, una canción debería ser eliminada y modificada únicamente por el autor de la canción. Tenemos un problema de seguridad, puesto que a las **peticiones GET / POST /cancion/modificar/:id y GET /cancion/eliminar puede acceder cualquiera.**

Vamos a crear en el **app.js** un router **"routerUsuarioAutor"** que: obtenga la id de la URL, busque la canción y verifique si el autor de esa canción coincide con el usuario en sesión. Las rutas a controlar por el router serán: **/cancion/modificar** y **/cancion/eliminar**.

Para obtener el parámetro id de la URL no podemos utilizar req.params.id. Esto se debe a que, la ruta que queremos controlar, pasa el parámetro directamente (cancion/eliminar/**valor**) en lugar de hacerlo con clave asociada (**?id=valor**).

El código de este router se añadirá en **app.js**, justo después de la aplicación del router **routerUsuarioSession**:

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

//routerUsuarioAutor
let routerUsuarioAutor = express.Router();
routerUsuarioAutor.use(function(req, res, next) {
  console.log("routerUsuarioAutor");
  let path = require('path');
  let id = path.basename(req.originalUrl);
  // Cuidado porque req.params no funciona
  // en el router si los params van en la URL.

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

//Aplicar routerUsuarioAutor
app.use("/cancion/modificar",routerUsuarioAutor);
app.use("/cancion/eliminar",routerUsuarioAutor);
```

Con esta comprobación, **obligamos a que únicamente el autor de la canción podrá modificar y/o eliminar sus canciones.**



Algunas aplicaciones, definen rutas específicas en las URLs para agruparlas según el nivel de autorización requerido, por ejemplo:

- /usuario/canciones/agregar
- /usuario/aut/cancion/modificar

Según el ejemplo anterior, cualquier usuario autenticado (**/usuario/**) puede agregar una canción. Sin embargo, además hay que ser el autor (**/aut/**) para modificarla. Empleando esta técnica se simplificaría el uso de los enrutadores, puesto que cada uno se aplica a una única URL. Por ejemplo: `app.use("/usuario/aut/", routerUsuarioAutor);`

Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-9.2- Gestión de colecciones- eliminar documento"*.

3 Redirección de peticiones

En este apartado, mejoraremos la navegabilidad de la aplicación incluyendo varias redirecciones que no especificamos en las prácticas anteriores. Por ejemplo, en **app.js** podríamos redireccionar la URL principal **GET /** a **/tienda**. Como se trata de la página de inicio es buena idea especificarlo en el propio **app.js**.

```
app.get('/', function (req, res) {  
  res.redirect('/tienda');  
})  
  
// lanzar el servidor  
app.listen(app.get('port'), function() {  
  console.log("Servidor activo");  
});
```

En **usuarios.js**, una vez identificado en la función **POST /identificarse**, vamos a redirigir al usuario a **/publicaciones**.

```
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: ");  
      res.redirect("/publicaciones");  
    }  
  })  
});
```



Para considerar el presente apartado como finalizado deben añadirse, como mínimo, las siguientes redirecciones:

- Canción agregada (**POST /cancion**) redirige a **GET /publicaciones**
- Canción modificada (**POST /cancion/modificar**) redirige **GET /publicaciones**
- Usuario registrado (**POST /usuario**) redirige a **GET /identificarse**

Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-9.3- Redirección de peticiones"*.

4 Mensajes y alertas

En este apartado, eliminaremos las respuestas de texto simple que tenemos en la aplicación (**res.send("texto")**). No obstante en muchos casos nos interesa enviar mensajes claros al usuario, especialmente el resultado de solicitar una operación. Por ejemplo:

- **/identificarse:** el usuario introduce sus datos incorrectamente.
- **/cancion/eliminar:** se ha eliminado la canción con éxito.
- Otros casos en los que queramos proporcionar feedback al usuario.

Como este tipo de mensajes van a ser comunes a toda la aplicación, los vamos a incluir en la plantilla de las vistas **base.html**. Incluiremos un **pequeño script jQuery que se ejecutará en el cliente y no tiene nada que ver con NodeJS**. El script obtendrá los parámetros **mensaje** y **tipoMensaje** de la URL, añadiendo el mensaje en **<div class="container">**.

El valor de **tipoMensaje** coincidirá con alguno de los tipos de alert que tiene Bootstrap (<https://getbootstrap.com/docs/3.3/components/#alerts>): **alert-success**, **alert-info** (por defecto), **alert-danger**, etc. Añadimos el siguiente código a **/views/base.html**:

```
<div class="container">
  <script>
    var mensaje = getUrlParameter('mensaje');
    var tipoMensaje = getUrlParameter('tipoMensaje');

    if ( mensaje != "" ){
      if (tipoMensaje == "" ){
        tipoMensaje = 'alert-info';
      }
      $( ".container" )
        .append("<div class='alert '+tipoMensaje+'>"+mensaje+" </div>");
    }

    function getUrlParameter(name) {
      name = name.replace(/[\/]/, '\\[\/]').replace(/[\\]/, '\\\\');
      var regex = new RegExp('[\\?&]' + name + '=([^\&]*)');
      var results = regex.exec(location.search);
      return results === null ? '' :
        decodeURIComponent(results[1].replace(/\+/g, ' '));
    };
  </script>

  <!-- Contenido -->
  {% block contenido_principal %}
  <!-- Posible contenido por defecto -->
  {% endblock %}
</div>
```



Nota: El objeto **URLSearchParams** de JavaScript permite obtener los parámetros GET, pero lamentablemente no funciona de forma correcta en algunos navegadores. En este caso optamos por implementar “manualmente” la función **getUrlParameter(name)**.

Abrimos el controlador **rusuarios.js** y modificamos la respuesta que se envía desde **POST /identificarse** cuando las credenciales del usuario son erróneas:

```
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.");
      res.redirect("/identificarse" +
        "?mensaje=Email o password incorrecto"+
        "&tipoMensaje=alert-danger ");
    } else {
      req.session.usuario = usuarios[0].email;
      res.redirect("/publicaciones");
    }
  });
});
```

Ejecutamos y comprobamos que sale el mensaje de error al iniciar sesión con datos erróneos:

Siguiendo este mismo enfoque, sustituimos los mensajes planos por redirecciones a URLs + mensajes. Modificamos las respuestas de **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 ");
      res.redirect("/registrarse?mensaje=Error al registrar usuario");
    } else {
      res.send('Usuario Insertado : ' + id);
      res.redirect("/identificarse?mensaje=Nuevo usuario registrado");
    }
  });
});
```




Nota: Podríamos aplicar los mensajes en muchas otras partes de la aplicación, ya es bueno darle al usuario feedback de las acciones que está realizando. Los mensajes de feedback no deben incluir información adicional que puedan comprometer la seguridad de nuestra aplicación.

Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-9.4- Mensajes y alertas"*.

5 Sistema de compra

A lo largo de este apartado, implementaremos el sistema de compra en la aplicación. Para comenzar, sustituimos en la vista de detalles de canción **/views/bcancion.html** el botón que muestra el precio, por un enlace a **/cancion/comprar/:id**

```
{% 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>
      <a class="btn btn-primary pull-right"
        href="/cancion/comprar/{{ cancion._id.toString() }}">{{ cancion.precio }}
      </a>
    </div>
  </div>
</div>
<!-- Cambiar el precio por "reproducir" si ya está comprada -->
```

Agregamos una nueva función **insertarCompra()** en **gestorBD.js** para añadir una compra que relacione al usuario con la canción comprada (es una relación N-N). Cada documento **compra** registrará el email del usuario y la id de la canción comprada. El código a añadir es:

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



En el controlador **rcanciones.js** agregamos la respuesta para **GET /cancion/comprar/:id**. Una vez agregada la compra, redirigiremos a **/GET /compras** (que implementaremos en el próximo paso):

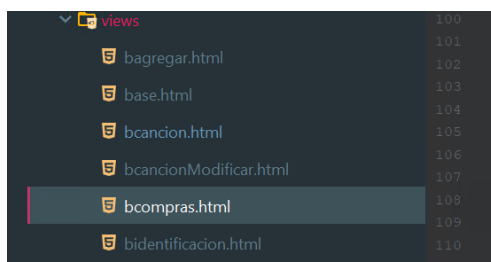
```
app.get('/cancion/comprar/:id', function (req, res) {
  let cancionId = gestorBD.mongo.ObjectId(req.params.id);
  let compra = {
    usuario : req.session.usuario,
    cancionId : cancionId
  }

  gestorBD.insertarCompra(compra ,function(idCompra){
    if ( idCompra == null ){
      res.send(respuesta);
    } else {
      res.redirect("/compras");
    }
  });
});
```

Implementamos **GET /compras** con el fin de que muestre todas las canciones compradas por el usuario. Primero debemos implementar la función **obtenerCompras ()** en **gestorBD.js**, que devolverá una lista de compras en base al criterio pasado como parámetro.

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

Movemos la vista **/public/bcompras.html** al directorio **/views/**.





La vista recibe la lista de canciones y las muestra, es muy similar a **bpublicaciones.html**, pero sin los botones de modificar y eliminar. Solo falta implementar **GET /compras**, siguiendo estos pasos:

1. Obtendremos primero todas las compras realizadas por el usuario que hay en sesión. Como cada compra tiene almacenada la Id de la canción comprada, podemos recuperar toda la información de la canción.
2. Guardamos las ids de todas las canciones compradas por el usuario en un array al que llamaremos **cancionesCompradasIds**. Invocamos a la función **obtenerCanciones** especificando como criterio que la id de la canción esté en ese array:

{ "_id" : { \$in: cancionesCompradasIds } }

Tras el último paso, ya tendremos todas las canciones compradas por el usuario y las pondremos a disposición de la vista **bcompras.html**. Añadimos a **rcanciones.js** el código:

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

  gestorBD.obtenerCompras(criterio ,function(compras){
    if (compras == null) {
      res.send("Error al listar ");
    } else {

      let cancionesCompradasIds = [];
      for(i=0; i < compras.length; i++){
        cancionesCompradasIds.push( compras[i].cancionId );
      }

      let criterio = { "_id" : { $in: cancionesCompradasIds } }
      gestorBD.obtenerCanciones(criterio ,function(canciones){
        let respuesta = swig.renderFile('views/bcompras.html',
          {
            canciones : canciones
          });
        res.send(respuesta);
      });
    }
  });
});
```

Revisamos los Routers declarados en **app.js** a para incluir restricciones en las nuevas URLs:

- **/cancion/comprar/:id** y **/compras** : comprobar que el usuario que accede está identificado en sesión.

```
//Aplicar routerUsuarioSession
app.use("/canciones/agregar",routerUsuarioSession);
app.use("/publicaciones",routerUsuarioSession);
app.use("/cancion/comprar",routerUsuarioSession);
app.use("/compras",routerUsuarioSession);
```



Ampliamos la lógica del **routerAudios** para que puedan acceder al fichero de audio **también los usuarios que han comprado la canción (además de los autores)**. Obtenemos el usuario y comprobamos su array de compras a ver si el ID de la canción se encuentra en él.

```
//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');

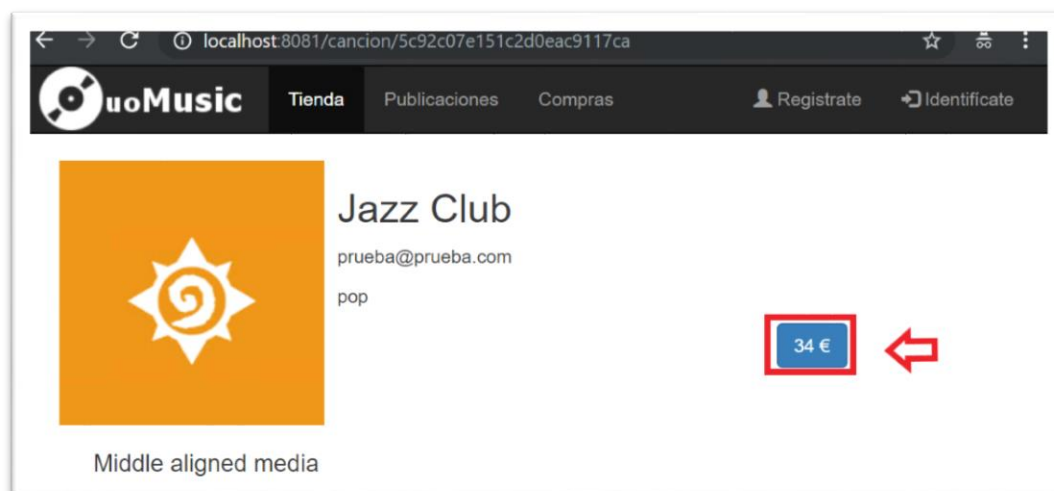
  gestorBD.obtenerCanciones(
    { "_id" : mongo.ObjectId(idCancion) }, function (canciones) {

      if( canciones[0].autor == req.session.usuario ){
        next();
      } else {
        //res.redirect("/tienda");
        let criterio = {
          usuario : req.session.usuario,
          cancionId : mongo.ObjectId(idCancion)
        };

        gestorBD.obtenerCompras(criterio ,function(compras){
          if (compras != null && compras.length > 0 ) {
            next();
          } else {
            res.redirect("/tienda");
          }
        });
      }
    }
  );
});
```

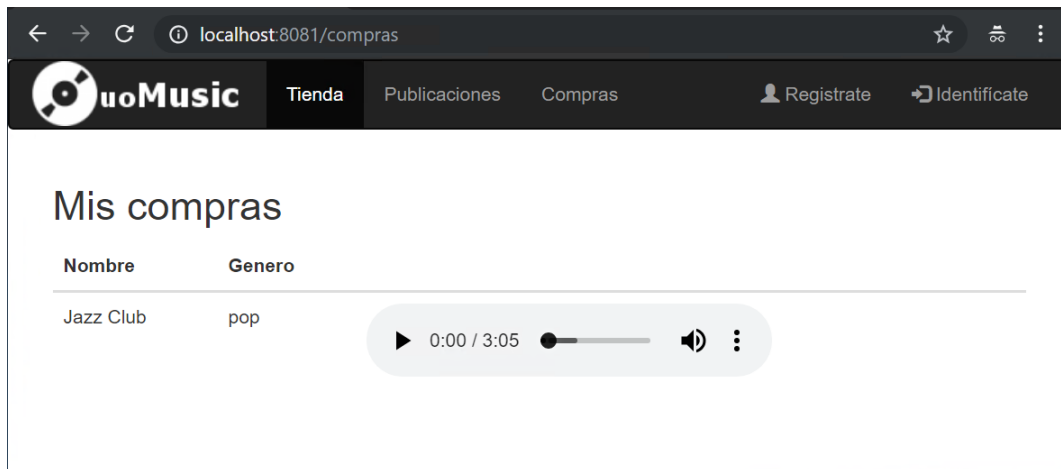
Nota: Después de un **next()** no podemos enviar otra respuesta o se producirá un error. Por tanto, una buena estrategia es incluir **return** después de los **next()**.

Guardamos los cambios y desde el detalle de una canción probamos a comprar una canción haciendo click sobre el precio.





Una vez comprada la canción, debería redirigirnos a la lista de canciones compradas del usuario, incluyendo la recién comprada.



Nota: Subir el código a GitHub en este punto. Commit Message → *“SDI-NodeJS-9.5- Sistema de compras”*.

6 Sistema de paginación

Listar todos los elementos de una colección cuando ésta tiene un elevado número de elementos, no es una buena práctica. Normalmente, las colecciones se suelen pagar, de tal forma que se muestren un número predeterminado de elementos por página. En este apartado vamos a crear un sistema de paginación desde cero, a pesar de que NodeJS tenga módulos específicos para ello, como **express-paginate** (<https://github.com/expressjs/express-paginate>).

En **gestorBD.js** implementamos la función **obtenerCancionesPg(pag)**, que devolverá las canciones correspondientes a una página concreta. Vamos a partir de **una configuración de 4 canciones por página**, por lo que: la página 1 tendrá las canciones 1 – 4, la página 2 las canciones 5 – 8 y así sucesivamente.

El código consistirá en obtener la colección de canciones, contar cuántas hay dentro del resultado (**función count**), saltarnos $(\text{NumPagina}-1)*4$ canciones en función del número de página solicitado (**función skip**) para finalmente obtener las 4 siguientes (**función limit**).

A la función de callback le enviamos:

1. La lista de canciones que deberían ir en la página indicada como parámetro.
2. El número total de canciones para que el sistema de paginación sepa cuántas páginas debe generar.

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



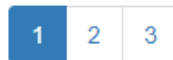
```
obtenerCancionesPg : function(criterio,pg,functionCallback){
    this.mongo.MongoClient.connect(this.app.get('db'), function(err, db) {
        if (err) {
            functionCallback(null);
        } else {
            let collection = db.collection('canciones');
            collection.count(function(err, count){

                collection.find(criterio).skip( (pg-1)*4 ).limit( 4 )
                    .toArray(function(err, canciones) {

                        if (err) {
                            functionCallback(null);
                        } else {
                            functionCallback(canciones, count);
                        }
                        db.close();
                    });
            });
        }
    });
};
```

En el controlador **rcanciones.js** modificamos el contenido de la función **GET /tienda**, para que reciba un parámetro GET con nombre **pg**. Este parámetro indicará la página a mostrar, por ejemplo, **tienda?pg=2**. Tenemos que tener en cuenta que es un **parámetro opcional** (por lo que si no existe, lo trataremos como valor 1) y que **su valor será un String** (debemos convertirlo con la función `parseInt`).

A la vista le vamos a enviar un array **"paginas"** cuyo contenido serán los números de página que deben mostrarse en el selector de página.



Normalmente no se suelen mostrar todas las páginas en el selector de página. En este caso, vamos a mostrar, respetando los límites de 0 y la última página: **desde 2 páginas antes de la actual, hasta 2 páginas tras la actual**. Al calcular el límite de la última página debemos tener **cuidado** con los decimales, **la división nos dice que para mostrar 5 canciones necesitamos 1.25 páginas, pero realmente necesitamos 2**.

Enviamos a la vista **bTienda.html** la lista de **canciones** (como antes pero limitada a 4 canciones), la lista de **números de página** para crear el selector de página y el número de página **actual** que utilizaremos para indicar visualmente al usuario donde está.

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

    let pg = parseInt(req.query.pg); // Es String !!!
    if ( req.query.pg == null ){ // Puede no venir el param
        pg = 1;
    }
});
```

```
gestorBD.obtenerCancionesPg(criterio, pg, function(canciones, total) {
    if (canciones == null) {
        res.send("Error al listar ");
    } else {
        let ultimaPg = total/4;
        if (total % 4 > 0) { // Sobran decimales
            ultimaPg = ultimaPg+1;
        }
        let paginas = []; // paginas mostrar
        for(let i = pg-2; i <= pg+2; i++){
            if (i > 0 && i <= ultimaPg){
                paginas.push(i);
            }
        }
        let respuesta = swig.renderFile('views/btienda.html',
        {
            canciones : canciones,
            paginas : paginas,
            actual : pg
        });
        res.send(respuesta);
    }
});
```

Modificaremos la parte final de la vista **bTienda.html** para recorrer la **lista de números de página** (son las que mostraremos en el selector de página) y añadimos un script para que incluya una clase CSS (active) a la página **actual**, lo que hará que se muestre resaltada.

```
<!-- Paginación mostrar la actual y 2 anteriores y dos siguientes -->
<div class="row text-center">
    <ul class="pagination">
        {% for pagina in paginas %}
        <li class="page-item" id="pi-{{ pagina }}">
            <a class="page-link" href="/tienda?pg={{ pagina }}" >{{ pagina }}</a>
        </li>
        {% endfor %}
        <script>
            $( "#pi-{{ actual }}" ).addClass("active");
        </script>
    </ul>
</div>
```

Para poder **acceder con jQuery a los elementos HTML** de manera sencilla, es importante utilizar el **atributo id**. En este caso hemos llamado **pi-1, pi- , etc**; a los elementos de la lista.

Guardamos los cambios, ejecutamos la aplicación y comprobamos que la paginación funciona.



Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-9.6- Sistema de paginación"*.



7 Manejo de errores en la aplicación

Por defecto y en algunos entornos de desarrollo, se deja que la aplicación lance excepciones. Cada vez que se lanza una excepción, se muestra la traza asociada y no se detiene la ejecución de la aplicación.

Sin embargo, **en entornos de producción, no es nada recomendable gestionar así las excepciones**. Vamos a comprobar que si accedemos a los detalles de una canción con un formato inválido de ID: <http://localhost:8081/cancion/RRRRRRRRRRRRRRRRRRRRRR>, se produce una excepción y la aplicación envía al cliente un código **Status 500**.

```
Error: Argument passed in must be a single String of 12 bytes or a string of 24 hex characters
    at new ObjectId (C:\Dev\workspaces\NodeApps\sdi-lab-node\node_modules\bson\lib\bson\objectid.js:57:11)
    at Function.ObjectId (C:\Dev\workspaces\NodeApps\sdi-lab-node\node_modules\bson\lib\bson\objectid.js:38:43)
```

Una forma de controlar las excepciones antes de que lleguen al cliente es interceptándolas, almacenarlas en un log y enviar al cliente una respuesta distinta. Puede ser desde una página de error personalizada, hasta un simple mensaje de error, tal y como vamos a hacer a continuación.

Añadimos la función básica de manejo de errores en el fichero **app.js**:

```
app.use( function (err, req, res, next ) {
  console.log("Error producido: " + err); //mostramos el error en consola
  if (! res.headersSent) {
    res.status(400);
    res.send("Recurso no disponible");
  }
});

app.listen(app.get('port'), function() {
  console.log("Servidor activo");
});
```

Nota: Los objetos `err` o `err.stack` nos proporcionan información sobre el error.

Accedemos de nuevo a la URL anterior para ver que al cliente se le muestra un mensaje de error, mientras que el error se está mostrando en la ejecución del servidor:

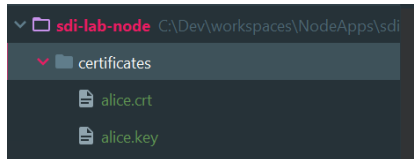
```
Error producido: Error: Argument passed in must be a single String of 12 bytes or a string of 24 hex characters
```




8 Implementando el protocolo HTTPS

En NodeJS, crear conexiones seguras entre cliente y servidor utilizando HTTPS es un proceso bastante sencillo.

En primer lugar, descargamos el fichero `certificates.zip` del Campus Virtual que contiene los certificados SSL **alice.crt** y **alice.key**. Creamos una **carpeta certificates** en nuestro proyecto y copiamos dentro los dos ficheros.



Seguidamente, abrimos el fichero principal **app.js** e incluimos dos nuevos módulos **fs (file system)** y **https**. No es necesario descargarlos puesto que están incluidos en el core de NodeJS.

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

let fs = require('fs');
let https = require('https');
```

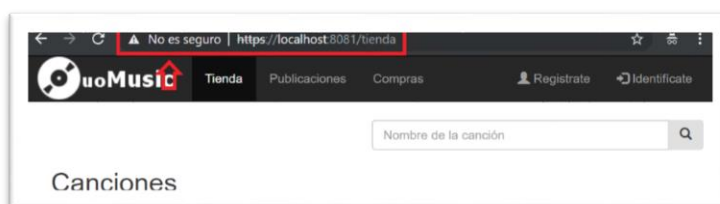
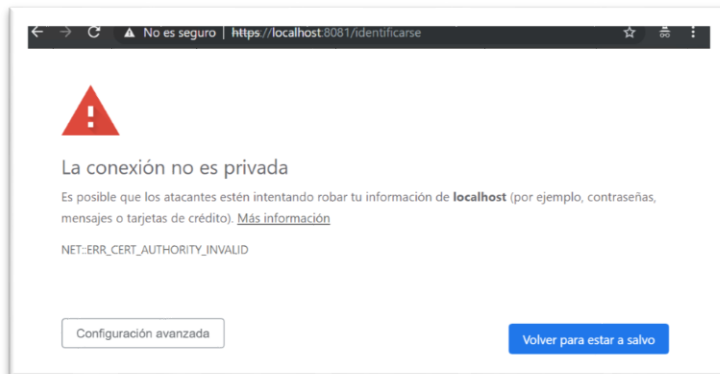
Modificamos la creación del servidor para utilizar https, indicándole donde está la clave y el certificado.

```
app.listen(app.get('port'), function() {
  console.log("Servidor activo");
});

https.createServer({
  key: fs.readFileSync('certificates/alice.key'),
  cert: fs.readFileSync('certificates/alice.crt')
}, app).listen(app.get('port'), function() {
  console.log("Servidor activo");
});
```

A partir de ahora, accederemos a la aplicación mediante <https://localhost:8081>. El navegador nos advertirá que el certificado de la página no es de confianza y **debemos confirmar una excepción de seguridad para el certificado**.


Básicamente, lo que nos dice el navegador es que, a pesar de que son certificados válidos, no los ha generado una **entidad certificadora confiable**.




Nota: Subir el código a GitHub en este punto. Commit Message → “SDI-NodeJS-9.7 y 9.8- Manejo de errores en la aplicación e implementando HTTPS”.

9 Resultado esperando el repositorio de GitHub


SDI-NodeJS-9.6- Sistema de paginación

 melsanchezsantillan committed 4 minutes ago


SDI-NodeJS-9.5- Sistema de compras

 melsanchezsantillan committed 7 minutes ago


SDI-NodeJS-9.4- Mensajes y alertas

 melsanchezsantillan committed 14 minutes ago

SDI-NodeJS-9.3- Redirección de peticiones

 melsanchezsantillan committed 16 minutes ago

SDI-NodeJS-9.2- Gestión de colecciones- eliminar documento

 melsanchezsantillan committed 25 minutes ago