Sistemas Distribuidos e Internet
Curso 2022/2023

Desarrollo de aplicaciones web con Node.js
Sesión - 9

Contenido

¡¡¡¡¡¡MUY IMPORTANTE!!!!!!

En este guion se ha usado como nombre de repositorio para todo el ejercicio **sdiIDGIT-lab-nodejs** . Sin embargo, cada alumno deberá usar como nombre de repositorio **sdix-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 STS: **trabajaremos sobre el proyecto del guion anterior.**

Colaborador invitado: **sdigithubuniovi**

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

nodemon[1] es una herramienta que reinicia automáticamente el servidor de la aplicación Node.js 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:

Para instalaciones de npm 5.2 o superior

```
npx nodemon www.js -e twig,js,json
```

Para instalaciones de npm anteriores a 5.2

```
nodemon .\bin\www -e twig,js,json
```

```
PS C:\Dev\Projects\SDI_projects\sdiIDGIT-lab-nodejs\musicstoreapp> nodemon .\bin\www -e twig, js, json
[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: twig,js,json
[nodemon] starting `node .\bin\www`
```

[1] https://www.npmjs.com/package/nodemon

A partir de ahora**, cada vez que modifiquemos y guardemos un fichero de la aplicación, cuya extensión sea .twig, .js, .json, el servidor se reiniciará automáticamente**. Cuando nodemon detecte un cambio en un fichero mostrará este mensaje:

```
[nodemon] starting `node .\bin\www`
[nodemon] restarting due to changes...
[nodemon] starting `node .\bin\www`
```

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, debemos primero parar el proceso lanzado con nodemon**.

Es posible que no os funcione desde una terminal del IDE. Probad a ejecutarlo desde una terminal abierta con CMD.

2 Eliminar canciones

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

Actualizar repositorio

Implementamos la función **deleteSong** en el fichero **repositories/songRepository.js**:

```
module.exports = {
  mongoClient: null,
  app: null,
  init: function (app, mongoClient) {
    this.mongoClient = mongoClient;
    this.app = app;
  },
  deleteSong: 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 result = await songsCollection.deleteOne(filter, options);
      return result;
    } catch (error) {
      throw (error);
    }
  }, updateSong: async function (newSong, filter, options) {...}
```

Actualizar controlador

A continuación, añadimos al controlador **routes/songs.js** una función que procese peticiones **GET /songs/delete/:id**. El parámetro id pertenecerá a la canción que queremos eliminar y se lo reenviaremos a **songsRepository.deleteSong().** El código para añadir al controlador es el siguiente:

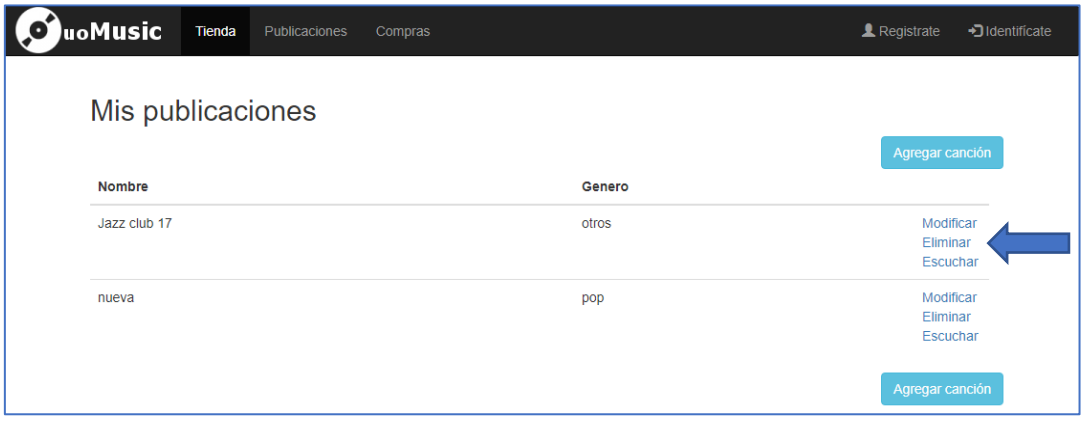
```
app.get('/songs/delete/:id', function (req, res) {
  let filter = {_id: ObjectId(req.params.id)};
  songsRepository.deleteSong(filter, {}).then(result => {
    if (result === null || result.deletedCount === 0) {
      res.send("No se ha podido eliminar el registro");
    } else {
      res.redirect("/publications");
    }
  }).catch(error => {
    res.send("Se ha producido un error al intentar eliminar la canción: " + error)
  });
})
```

Actualizar vista

Verificamos que el enlace para borrar canciones esté correcto en el fichero **publications.twig**

```
...
<a href="/songs/delete/{{ song._id }}">Eliminar</a> <br>
...
```

Sí accedemos a la aplicación podemos observar que las canciones se eliminan correctamente, pulsando en el enlace eliminar.



Control de acceso vía router

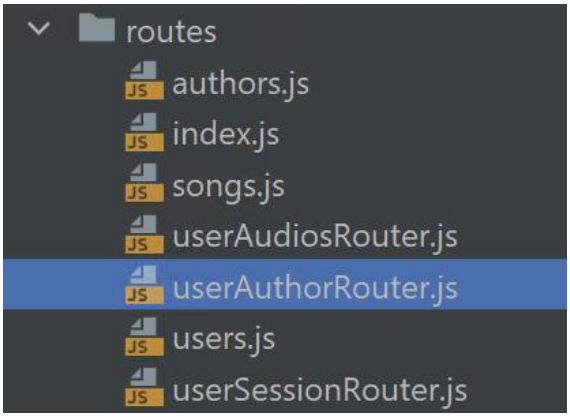
Una canción debería ser eliminada y modificada únicamente por el ***autor/propietario*** de la canción. Actualmente, tenemos un problema de seguridad, puesto que a las **peticiones GET y POST** a **/songs/edit/:id y /songs/delete/:id puede acceder cualquiera.**

Vamos a implementar un router "**userAuthorRouter"** que obtenga la id de la canción de la URL, busque la canción y verifique si el autor de esa canción coincide con el usuario en sesión. Las rutas a interceptar por el router serán: **/songs/edit** y **/songs/delete.**

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 (***songs/delete/valor***) en lugar de hacerlo con clave asociada (**?id=valor**).

Implementar el router userAuthorRouter

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




```
const express = require('express');
const path = require("path");
const {ObjectId} = require("mongodb");
const songsRepository = require("../repositories/songsRepository");
const userAuthorRouter = express.Router();
userAuthorRouter.use(function (req, res, next) {
  console.log("userAuthorRouter");
  let songId = path.basename(req.originalUrl);
  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 = userAuthorRouter;
```

Luego, incluimos el router el fichero principal **app.js**, justo después del router ***userSessionRouter***:

```
app.use("/shop/",userSessionRouter)

const userAuthorRouter = require('./routes/userAuthorRouter');
app.use("/songs/edit",userAuthorRouter);
app.use("/songs/delete",userAuthorRouter);
```

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:

- /users/songs/add
- / users /auth/songs/edit

Según el ejemplo anterior, cualquier usuario autenticado **(/users/**) puede agregar una canción. Sin embargo, además hay que ser el autor (**/auth/**) 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("/users/auth/", userAuthorRouter);***

Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-9.1-Gestión de colecciones, eliminar documento."

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

"SDI-2223-101-9.1-Gestión de colecciones, eliminar documento."

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

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 **routes/index.js** podríamos redireccionar a la URL principal **GET /** a **/shop** en lugar de index **(**actualmente no tiene ningún contenido relevante).

```
let express = require('express');
let router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  //res.render('index', { title: 'Express' });
  res.redirect('/shop');
});

module.exports = router;
```

En **routes/users.js**, una vez identificado en la función **POST /users/login**, vamos a redirigir al usuario a **/publications**, en caso contrario, podríamos mantenernos en la página actual y por ejemplo mostrar un mensaje de error si el login ha fallado (más adelante hablaremos de la gestión de errores).

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

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

- Canción agregada (**POST /songs/add**) redirige a **GET /publications**
- Canción modificada (**POST /songs/edit** redirige a **GET /publications**
- Usuario registrado (**POST / users/signup**) redirige a **GET /users/login**

Por otro lado, habría que revisar el menú de navegación para que las opciones funcionen correctamente.

Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-9.2-Redirección de peticiones."

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

"SDI-2223-101-9.2-Redirección de peticiones."

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

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 tras solicitar una operación. Por ejemplo:

- **/users/login:** el usuario introduce sus datos incorrectamente.
- **/songs/delete:** se ha eliminar la canción con éxito.
- Otros casos en el 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 **layout.twig.** Para ello, incluiremos un **pequeño script jQuery que se ejecutará en el cliente. Este script, que no tiene nada que ver con Node.js**, obtendrá los parámetros **message** y **messageType** de la URL, añadiendo el mensaje en ** .**

El valor de **messageType** 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.**

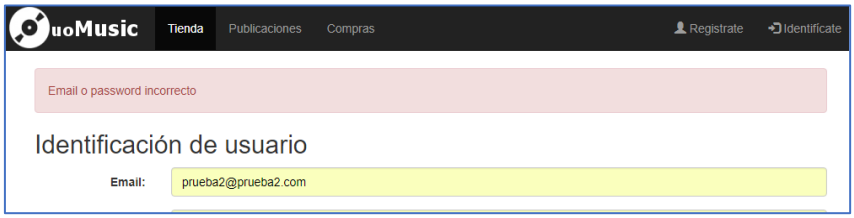
Añadimos el siguiente código a **/views/layout.twig:**

```
  <!-- Contenido -->
  {% block main_container %}
    <!-- Posible contenido por defecto -->
  {% endblock %}
<div class="container">
  <script>
    let params = new URLSearchParams(location.search);
    let message = params.get("message");
    let messageType = params.get("messageType");

    if (message !== null && message !== "") {
      if (messageType === "") {
        messageType = 'alert-info';
      }
      $(".container")
            .append("<div class='alert " + messageType + "'>" + message + " </div>");
    }
  </script>

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

En el controlador **routes/users.js** modificamos la respuesta que se envía desde **POST /users/login** cuando las credenciales del usuario son erróneas o se ha producido algún error:

```
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");
      res.redirect("/users/login" +
        "?message=Email o password incorrecto"+
        "&messageType=alert-danger ");
    } else {
      req.session.user = user.email;
      //res.send("Usuario identificado correctamente: " + user.email
      res.redirect("/publications");
    }
  }).catch(error => {
    req.session.user = null;
    //res.send("Se ha producido un error al buscar el usuario: " + error)
    res.redirect("/users/login" +
      "?message=Se ha producido un error al buscar el usuario"+
      "&messageType=alert-danger ");
  })
})
```

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



Siguiendo este mismo enfoque, sustituimos los mensaje planos por redirecciones a URLs + mensajes. Modificamos las respuestas de **POST /users/signup:**

```
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
  }
  usersRepository.insertUser(user).then(userId => {
    res.redirect("/users/login" + '?message=Nuevo usuario registrado.'+
      "&messageType=alert-info");
  }).catch(error => {
    res.redirect(" /users/signup" +
      "?message=Se ha producido un error al registrar el usuario." +
      "&messageType=alert-danger")
  });
});
```

Nota: Podríamos aplicar los mensajes en otras partes de la aplicación, ya que 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: Incluir el siguiente Commit Message ->

"SDI-IDGIT-9.3-Mensajes y alertas."

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

"SDI-2223-101-9.3-Mensajes y alertas."

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

5 Sistema de compra

A lo largo de este apartado, implementaremos el sistema de compras en la aplicación. Para comenzar, sustituimos en la vista de detalles de canción **/views/songs/song.twig** el botón que muestra el precio, por un enlace a **/songs/buy/:id**

```
{% block main_container %}
  <div class="row">
    <div class="media col-xs-10">
      <div class="media-left media-middle">
        <img class="media-object" src="/covers/{{ song._id }}.png">
      </div>
      <div class="media-body">
        <h2>{{ song.title }}</h2>
        <p>{{ song.kind }}</p>
        <p>{{ song.author }}</p>
```

```
        <button type="button" class="btn btn-primary pull-right">{{ song.price }} €</button>
        <a class="btn btn-primary pull-right"
          href="/songs/buy/{{ song._id }}">{{ song.price }} €</a>
      </div>
    </div>
  </div>
{% endblock %}
```

Actualizar repositorio

Añadimos una nueva función **buySong()** en **repositories/songsRepository.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 = {
  mongoClient: null,
  app: null,
  init: function (app, mongoClient) {
    this.mongoClient = mongoClient;
    this.app = app;
  },
  buySong: function (shop, callbackFunction) {
    this.mongoClient.connect(this.app.get('connectionStrings'), function (err, dbClient) {
      if (err) {
        callbackFunction(null)
      } else {
        const database = dbClient.db("musicStore");
        const collectionName = 'purchases';
        const purchasesCollection = database.collection(collectionName);
        purchasesCollection.insertOne(shop)
          .then(result => callbackFunction(result.insertedId))
          .then(() => dbClient.close())
          .catch(err => callbackFunction({error: err.message}));
      }
    });
  },
```

La colección purchases de MongoDB almacenará los documentos con la siguiente estructura:

```
_id: ObjectId("62482a4d95812f618eaa8c7d")
user: "prueba1@prueba1.com"
songId: ObjectId("623c305df4d33e7370efa172")
```

Nota: Otra opción podría ser guardar en la colección el userId (id de mongo) en lugar del email.

A continuación implementamos la función **getPurchases(),** que devolverá una lista de compras en base al criterio/filtro pasado como parámetro:

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

Actualizar controlador

En el controlador **routes/songs.js** añadimos la respuesta para **GET /songs/ buy/:id.** Una vez realizada la compra, redirigiremos a **/GET /purchases** (que implementaremos en el próximo paso):

```
app.get('/songs/buy/:id', function (req, res) {
  let songId = ObjectId(req.params.id);
  let shop = {
    user: req.session.user,
    songId: songId
  }
  songsRepository.buySong(shop, function (shopId) {
    if (shopId == null) {
      res.send("Error al realizar la compra");
    } else {
      res.redirect("/purchases");
    }
  })
});
```

Ahora implementamos **GET /purchases** con el fin de que muestre todas las canciones compradas por el usuario. Los pasos que se realizan en esta función son los siguientes:

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 **purchasedSongsIds**. Invocamos a la función getSongs() **especificando como criterio/filtro que la id de la canción esté en ese array**:

{"_id" : { $in: purchasedSongsId } }

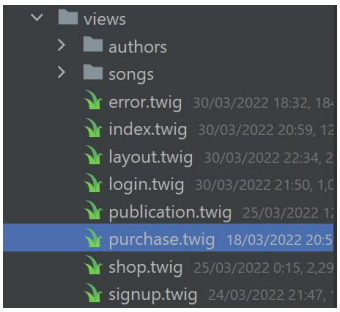
Tras el último paso, ya tendremos todas las canciones compradas por el usuario y las pondremos a disposición de la vista **purchase.twig.** Añadimos el siguiente endpoint en **routes/songs.js:**

```
app.get('/purchases', function (req, res) {
  let filter = {user: req.session.user};
```

```
  let options = {projection: {_id: 0, songId: 1}};
  songsRepository.getPurchases(filter, options).then(purchasedIds => {
    let purchasedSongs = [];
    for (let i = 0; i < purchasedIds.length; i++) {
      purchasedSongs.push(purchasedIds[i].songId)
    }
    let filter = {"_id": {$in: purchasedSongs}};
    let options = {sort: {title: 1}};
    songsRepository.getSongs(filter, options).then(songs => {
      res.render("purchase.twig", {songs: songs});
    }).catch(error => {
      res.send("Se ha producido un error al listar las publicaciones del usuario: " + error)
    });
  }).catch(error => {
    res.send("Se ha producido un error al listar las canciones del usuario " + error)
  });
})
```

Actualizar vistas

Movemos y renombramos la vista **/public/buy.html** al directorio **/views/purchase.twig**.



La vista recibe la lista de canciones y las muestra, es muy similar a **publications.twig**, pero sin los botones de modificar y eliminar. Copiamos el siguiente código en **purchases.twig**

```
{% extends "layout.twig" %}
{% block title %} Mis Compras {% endblock %}
{% block main_container %}
  <h2>Mis compras</h2>
  <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>
              <audio controls>
                <source src="/audios/{{ song._id }}.mp3" type="audio/mpeg">
              </audio>
            </td>
          </tr>
        {% endfor %}
        </tbody>
      </table>
    </div>
{% endblock %}
```

En el fichero **layout.twig** cambiamos el siguiente enlace para que desde la menú de compras podamos acceder directamente a las compras realizadas:

```
<!-- <li id="mybuy"><a href="/buy">Compras</a></li> -->
<li id="mybuy"><a href="/purchases">Compras</a></li>
```

Actualizar los routers (interceptores)

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

- **/songs/buy/:id y /purchases**: comprobar que el usuario que accede está identificado en sesión.

```
app.use("/songs/add",userSessionRouter);
app.use("/publications",userSessionRouter);
app.use("/songs/buy",userSessionRouter);
app.use("/purchases",userSessionRouter);
app.use("/audios/",userAudiosRouter);
app.use("/shop/",userSessionRouter)
```

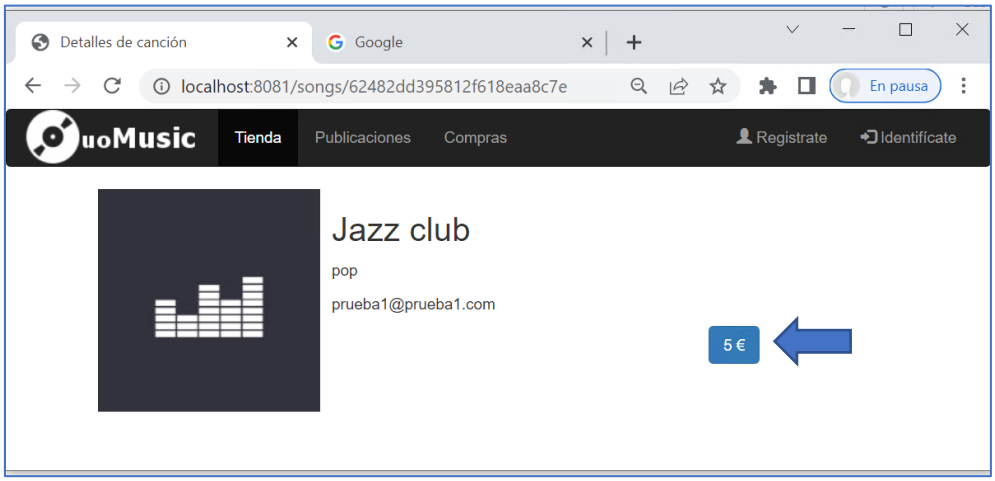
A continuación, ampliamos la lógica del **userAudiosRouter** para que puedan acceder al fichero de audio **también los usuarios que han comprado la canción (además de los autores).** Obtenemos el usuario y comprobamos su array de compras a ver si el ID de la canción se encuentra en él.

```
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 {
```

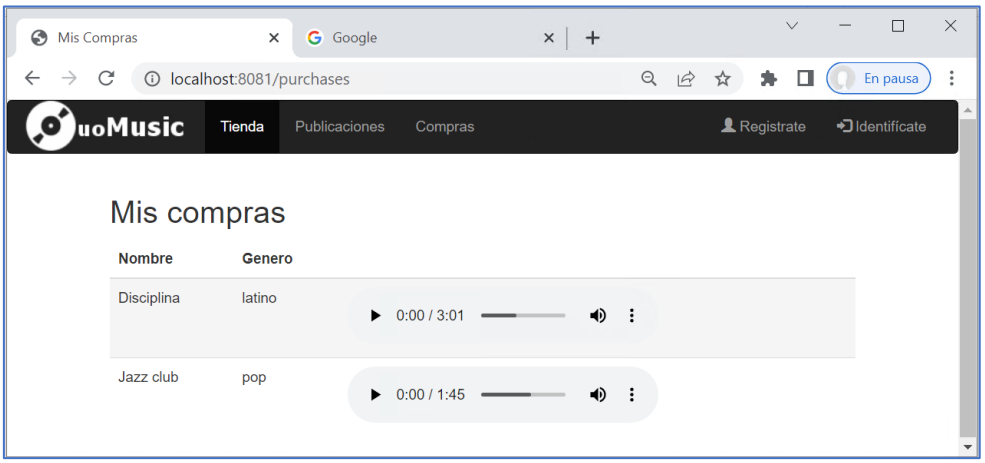
```
        //res.redirect("/shop");
        let filter = {user: req.session.user, songId: ObjectId(songId)};
        let options = {projection: {_id: 0, songId: 1}};
        songsRepository.getPurchases(filter, options).then(purchasedIds => {
          if (purchasedIds !== null && purchasedIds.length > 0) {
            next();
          } else {
            res.redirect("/shop");
          }
        }).catch(error => {
          res.redirect("/shop");
        })
    }
  }).catch(error => {
    res.redirect("/shop");
  });
});
module.exports = userAudiosRouter;
```

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 clic sobre el precio.



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



Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-9.4-Sistema de compras."

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

"SDI-2223-101-9.4-Sistema de compras."

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

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 paginar 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 Node.js tenga módulos específicos para ello, como **express-paginate** (https://github.com/expressjs/express-paginate).

Modificar Repositorio

En **repositories/songsRepository.js** implementamos la función **getSongsPg(page),** 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/método count()**), saltarnos (NumPagina-1)*4 canciones en función del número de página solicitado (**función/método skip()**) para finalmente obtener las 4 siguientes (**función/método limit()**).

La función devolverá un objeto conteniendo:

1. La lista de canciones que deberían ir en la página indicada como parámetro.

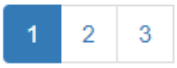
2. El número total de canciones, para que el sistema de paginación sepa cuántas páginas debe generar.

```
getSongsPg: async function (filter, options, page) {
   try {
      const limit = 4;
      const client = await this.mongoClient.connect(this.app.get('connectionStrings'));
      const database = client.db("musicStore");
      const collectionName = 'songs';
      const songsCollection = database.collection(collectionName);
      const songsCollectionCount = await songsCollection.count();
      const cursor = songsCollection.find(filter, options).skip((page - 1) * limit).limit(limit)
      const songs = await cursor.toArray();
      const result = {songs: songs, total: songsCollectionCount};
      return result;
   } catch (error) {
      throw (error);
   }
}, getSongs: async function (filter, options) {..}
```

Modificar controlador

En el controlador **routes/songs.js** modificamos el contenido de la función **GET /shop**, para que reciba un párametro GET con el nombre **page**. Este parámetro indicará la página a mostrar, por ejemplo, **shop/?page=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 **shop.twig** 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('/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": {$regex: ".*" + req.query.search + ".*"}};
   }
   let page = parseInt(req.query.page); // Es String !!!
   if (typeof req.query.page === "undefined" || req.query.page === null || req.query.page === "0") { //
Puede no venir el param
      page = 1;
   }
   songsRepository.getSongsPg(filter, options, page).then(result => {
      let lastPage = result.total / 4;
      if (result.total % 4 > 0) { // Sobran decimales
         lastPage = lastPage + 1;
      }
      let pages = []; // paginas mostrar
      for (let i = page - 2; i <= page + 2; i++) {
         if (i > 0 && i <= lastPage) {
            pages.push(i);
         }
      }
      let response = {
         songs: result.songs,
         pages: pages,
         currentPage: page
      }
      res.render("shop.twig", response);
   }).catch(error => {
      res.send("Se ha producido un error al listar las canciones del usuario " + error)
   });
})
```

Modificaremos la parte final de la vista **shop.twig** 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.

```
<div class="row text-center">
   <ul class="pagination">
      {% for page in pages %}
         <li class="page-item" id="pi-{{ page }}">
            <a class="page-link" href="/shop/?page={{ page }}"> {{ page }} </a>
         </li>
      {% endfor %}
      <script>
         $("#pi-" + "{{ currentPage }}").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-N, etc;** a los elementos de la lista.

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

Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-9.5-Sistema de paginación."

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

"SDI-2223-101-9.5-Sistema de paginación"

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

7 Manejo de errores en la aplicación

Por defecto, 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**. Por ejemplo, si accedemos a los detalles de una canción con un formato inválido de ID: http://localhost:8081/songs/RRRRRRRRRRRRRRRRRRRR, se podría producir 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.

Express, por defecto, incluye una función básica de manejo de errores en el fichero **app.js**. Para ver cómo funciona, muestra por consola el error que se ha generado, añadiendo la siguiente línea en la función:

```
app.use(function (err, req, res, next) {
  // set locals, only providing error in development
  console.log("Se ha producido un error " + err);
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};
  // render the error page
  res.status(err.status || 500);
  res.render('error');
});
```

Nota: el objeto err nos proporcionan información sobre el error.

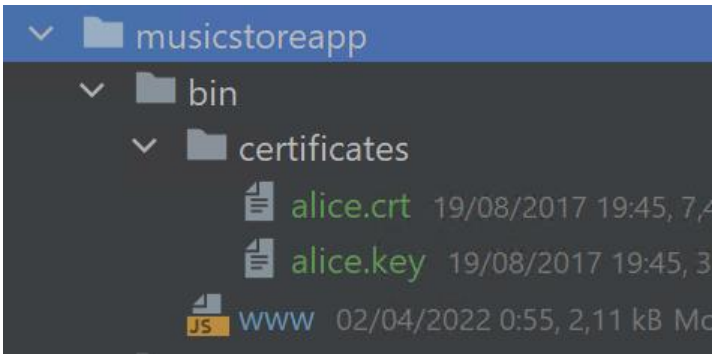
Accedemos de nuevo a la URL anterior para ver que al cliente se le muestra un mensaje de error, mientras que el error se está mostrando en la ejecución del servidor:

```
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 Node.js, crear conexiones seguras entre cliente y servidor utilizando HTTPS es un proceso bastante sencillo.

En primer lugar, descargamos el fichero certificates.zip de la carpeta compartida. El zip contiene los certificados SSL **alice.crt** y **alice.key**. Creamos una **carpeta certificates dentro de la carpeta bin** del proyecto y copiamos dentro los dos ficheros.



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

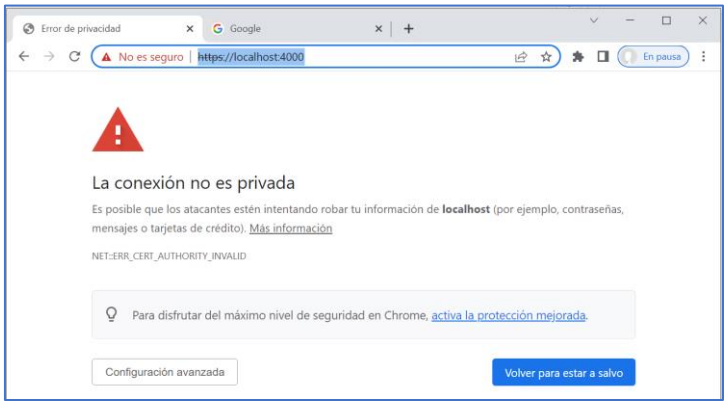
```
let app = require('../app');
let debug = require('debug')('musicstoreapp:server');
let http = require('http');

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

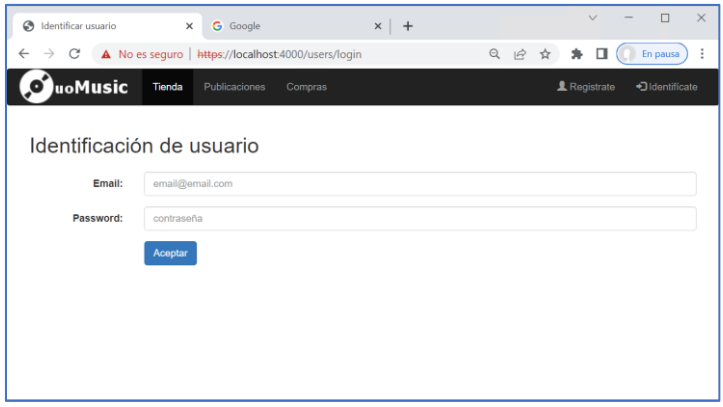
Modificamos la creación del servidor para utilizar https, indicándole donde está la clave y el certificado.

```
//var server = http.createServer(app);
let path = require('path');
let privateKey  = fs.readFileSync(path.join(__dirname,'certificates/alice.key'), 'utf8');
let certificate = fs.readFileSync(path.join(__dirname,'certificates/alice.crt'), 'utf8');
let credentials = {key: privateKey, cert: certificate}

let server = http.createServer(app);
let httpsServer = https.createServer(credentials, app);
httpsServer.listen(4000);

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

A partir de ahora, podemos **acceder a la aplicación empleando tanto el *protocolo HTTP como el HTTPS***. Según la elección, accederemos a la aplicación mediante https://localhost:4000 o http://localhost:8081. Cuando utilicemos ***HTTPS***, el navegador nos advertirá de 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: Incluir el siguiente Commit Message ->

"SDI-IDGIT-9.6-Manejo de errores y HTTPS."

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

"SDI-2223-101-9.6-Manejo de errores y HTTPS."

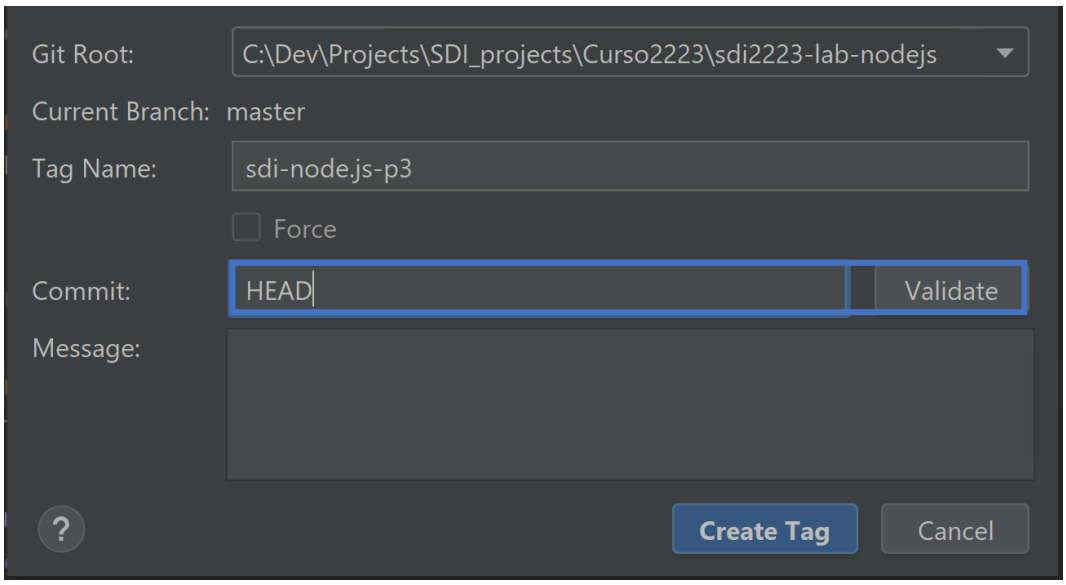
(No olvides incluir los guiones y NO incluyas BLANCOS al principio o al final, ni las comillas)

9 Etiquetar el proyecto

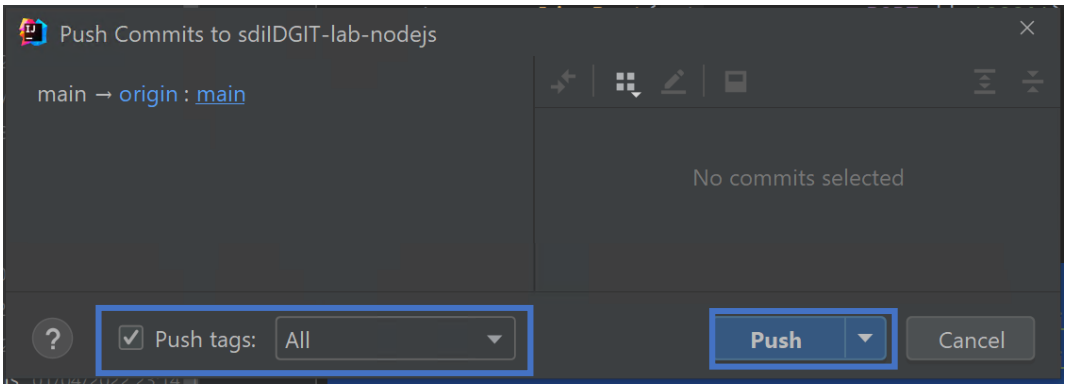
Nota: Etiquetar el proyecto en este punto con la siguiente etiqueta → **sdi-node.js-p3.**

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 el IDE e ir a la opción **Git → New Tag…**

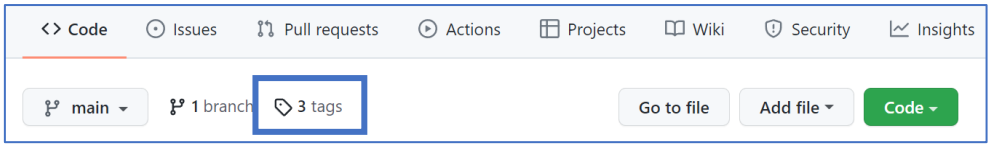
En la siguiente ventana indicamos el nombre de la etiqueta(**sdi-node-p2)**, un mensaje que indique la funcionalidad que se está etiquetado en este punto. Además**, añadimos el commit actual para crear la etiqueta escribiendo HEAD**. **Pulsamos el botón "Create Tag".**



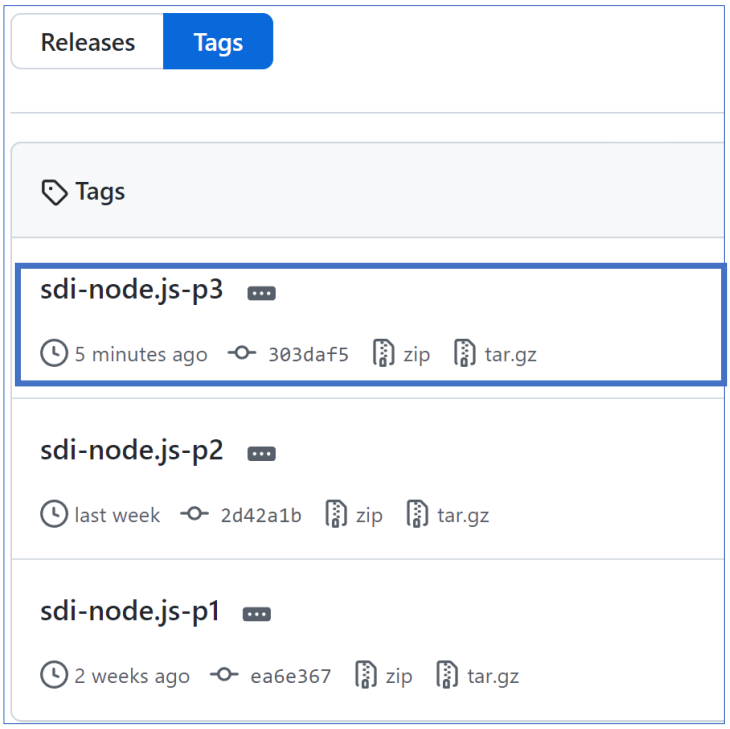
A continuación, vamos a hacer un push a la etiqueta: **Clic derecho en el proyecto → Git → Push…** En el cuadro que aparece, **seleccionamos Push tags: All** y hacemos clic en el botón **Push**.



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:



10 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-2223-101-9.1-Gestión de colecciones, eliminar documento.**
- ⇒ **SDI-2223-101-9.2-Redirección de peticiones.**
- ⇒ **SDI-2223-101-9.3-Mensajes y alertas.**
- ⇒ **SDI-2223-101-9.4-Sistema de compras.**
- ⇒ **SDI-2223-101-9.5-Sistema de paginación**
- ⇒ **SDI-2223-101-9.6-Manejo de errores y HTTPS.**