Sistemas Distribuidos e Internet
Desarrollo de aplicaciones web con Node.js (Servicio web REST)

Sesión - 10
Curso 2022/ 2023

Contenido

¡¡¡¡¡¡MUY IMPORTANTE!!!!!!

En este guion se ha usado como nombre de repositorio para todo el ejercicio **sdiIDGIT-lab-node.js**. Sin embargo, cada alumno deberá usar como nombre de repositorio **sdix-lab-node.js**, 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**

Introducción

A lo largo de esta práctica, **implementaremos y consumiremos servicios web REST desarrollados con Node.js**. La práctica se divide en cuatro partes:

1. Crear una **API de servicios web REST sobre la aplicación *tienda de música* (music_store)**. Esta API nos permitirá gestionar las canciones empleando una versión parcial de la entidad canción (sin fichero MP3 ni portada).
2. Implementar una **aplicación basada en jQuery-Ajax** que definirá una **interfaz de usuario y consumirá los servicios web REST** anteriormente implementados.
3. **Consumir un servicio web externo desde la aplicación *tienda de música***.
4. Una **aplicación Java que consumirá un servicio web REST** desarrollado en Node.js.

Parte 1 – Implementación de una API REST

Las peticiones y los servicios web

Cuando implementemos un servicio web, para cada petición devolveremos:

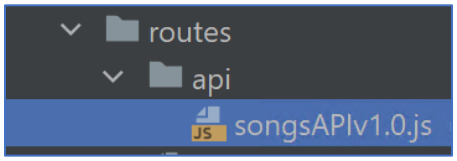
1. **El código de respuesta estándar** adecuado mediante **res.status**(código). Más información sobre códigos de estado de respuesta HTTP: https://developer.mozilla.org/es/docs/Web/HTTP/Status

2. **Una respuesta en un formato** que pueda ser fácilmente procesado por una aplicación, como es el caso de: **JSON, XML, YAML,** etcétera.

En este caso la gestión de las respuestas va a ser bastante directa, ya que estamos utilizando una base de datos que ya tiene todos sus datos en JSON. Si no fuera así, deberíamos aplicar obligatoriamente **JSON.stringify** para transformar los datos JSON a un objeto de tipo String. Con **res.json** también podemos transformar objetos String a formato JSON.

Gestión del recurso canción (CRUD song)

En **la carpeta routes, creamos un nuevo fichero songsAPIv1.0.js dentro de una subcarpeta que llamaremos "api"** y en el que implementaremos una API REST para gestionar las canciones de nuestra aplicación (operaciones CRUD).



Obtener lista de canciones - Método HTTP GET

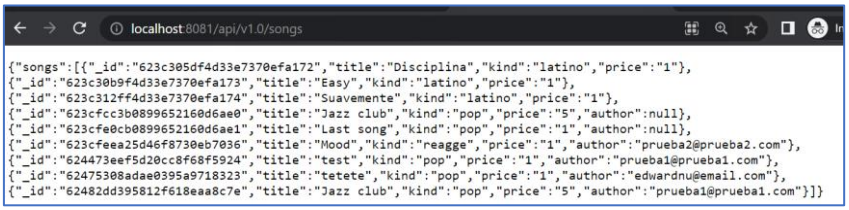
A través de una petición **GET** a **/api/v1.0/songs/** obtendremos una lista con todas las canciones de la aplicación, en formato JSON. Definimos el módulo y el método HTTP get, como muestra el siguiente código:

```
const {ObjectId} = require("mongodb");
module.exports = function (app, songsRepository) {
  app.get("/api/v1.0/songs", function (req, res) {
    let filter = {};
    let options = {};
    songsRepository.getSongs(filter, options).then(songs => {
      res.status(200);
      res.send({songs: songs})
    }).catch(error => {
      res.status(500);
      res.json({ error: "Se ha producido un error al recuperar las canciones." })
    });
  });
}
```

A continuación, incluimos el nuevo controlador en **app.js**

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

require("./routes/api/songsAPIv1.0.js")(app, songsRepository);
```

A lo largo de este guion, se presupone que la práctica anterior está completada y, por tanto, todos los enlaces se proporcionan a través de HTTPS por el puerto 4000 o HTTP por el puerto 8081. A continuación, probaremos el servicio realizando la petición GET a través del navegador: https://localhost:4000/api/v1.0/songs o http://localhost:8081/api/v1.0/songs.



HATEOAS (Hypermedia as the Engine of Application State): dentro de lo posible, los clientes tienen que poder ir explorando los recursos de la aplicación. Cuando desde un recurso se haga referencia a otro, se deben utilizar sus identificadores y, a ser posible, un enlace directo para explorar ese recurso. Por ejemplo**, la opción 2 sería mucho mejor que la 1**.

1. "author": prueba@prueba2.com
2. "author": /users/prueba2@prueba2.com

Obtener una canción por su ID - Método HTTP GET

La petición **GET** a **/api/v1.0/songs/:id** debería devolver la canción con el id correspondiente. Para lograr este objetivo, añadimos el siguiente endpoint al módulo:

```
app.get("/api/v1.0/songs/:id", function (req, res) {
  try {
    let songId = ObjectId(req.params.id)
    let filter = {_id: songId};
    let options = {};
    songsRepository.findSong(filter, options).then(song => {
      if (song === null) {
        res.status(404);
        res.json({error: "ID inválido o no existe"})
      } else {
        res.status(200);
        res.json({song: song})
      }
    }).catch(error => {
      res.status(500);
      res.json({error: "Se ha producido un error a recuperar la canción."})
    });
  } catch (e) {
    res.status(500);
    res.json({error: "Se ha producido un error :" + e})
  }
});
```

Eliminar una canción por su ID - Método HTTP DELETE

Para eliminar una canción, el servicio utilizará el método HTTP DELETE. La petición será del estilo **api/v1.0/songs/:id**. El resto de la petición será casi idéntica a la anterior.

```
app.delete('/api/v1.0/songs/:id', function (req, res) {
  try {
    let songId = ObjectId(req.params.id)
    let filter = {_id: songId}
    songsRepository.deleteSong(filter, {}).then(result => {
      if (result === null || result.deletedCount === 0) {
        res.status(404);
        res.json({error: "ID inválido o no existe, no se ha borrado el registro."});
      } else {
        res.status(200);
        res.send(JSON.stringify(result));
      }
    }).catch(error => {
      res.status(500);
      res.json({error: "Se ha producido un error al eliminar la canción."})
    });
  } catch (e) {
    res.status(500);
    res.json({error: "Se ha producido un error, revise que el ID sea válido."})
  }
});
```

Nota: Incluimos el código en un **bloque try/catch()**, para que si ocurre algún error inesperado, **no redireccione al router** que gestiona todos los errores ocurridos en la aplicación. **Este router está definido en el app.js como una función** (se puede encontrar buscando el comentario //error handler).

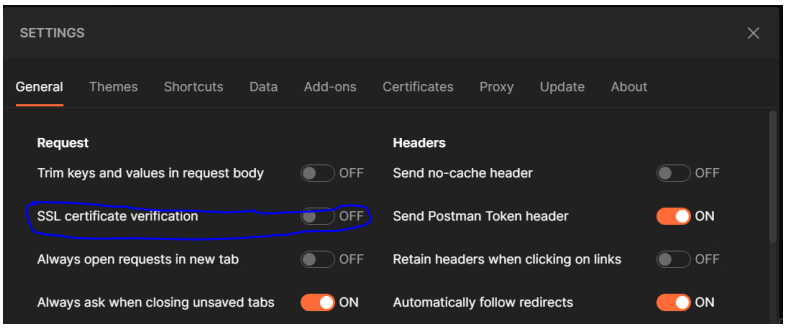
Mediante este enfoque, **si enviamos un ID inválido y queremos convertirlo en un ObjectID**: https://localhost:4000/api/v1.0/songs/623c30b9 obtendríamos el error: ***"Argument passed in must be a string of 12 bytes or a string of 24 hex characters or an integer"***

Para **probar peticiones de APIs REST tipo: DELETE, POST, PUT o PATCH;** podemos utilizar la aplicación **Postman.** Existen otras herramientas similares[1], por ejemplo, SoapUI, APIGee. **También sería posible emplear cURL** tal y como se describe en el seminario de HTTP.

Descargaremos POSTMAN desde: https://www.postman.com/downloads/. Tras finalizar la instalación, ya podemos enviar peticiones a nuestra API. Cabe destacar que **NO ES NECESARIO REGISTRARSE** para utilizar la aplicación.

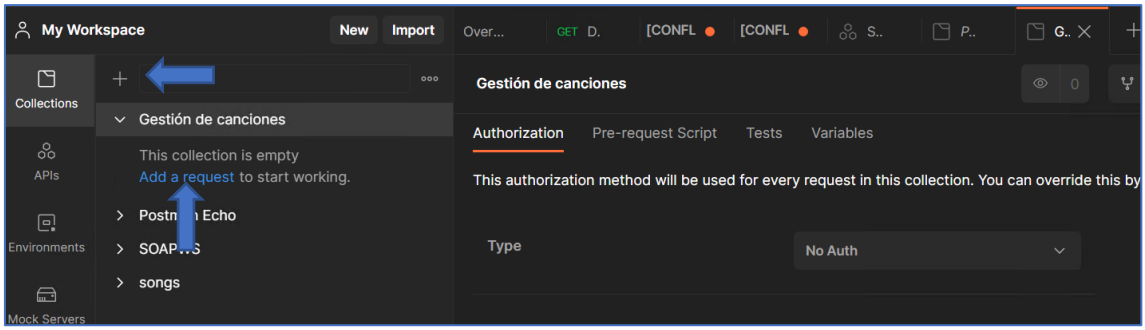
[1] Top 5 herramientas de prueba API REST 2021: alicealdaine.medium.com

A continuación, vamos a desactivar la verificación de SSL en la configuración: **File | Settings**:



En primer lugar, vamos a probar a realizar una **petición DELETE a la API utilizando la id de una canción que exista en la base de datos**.

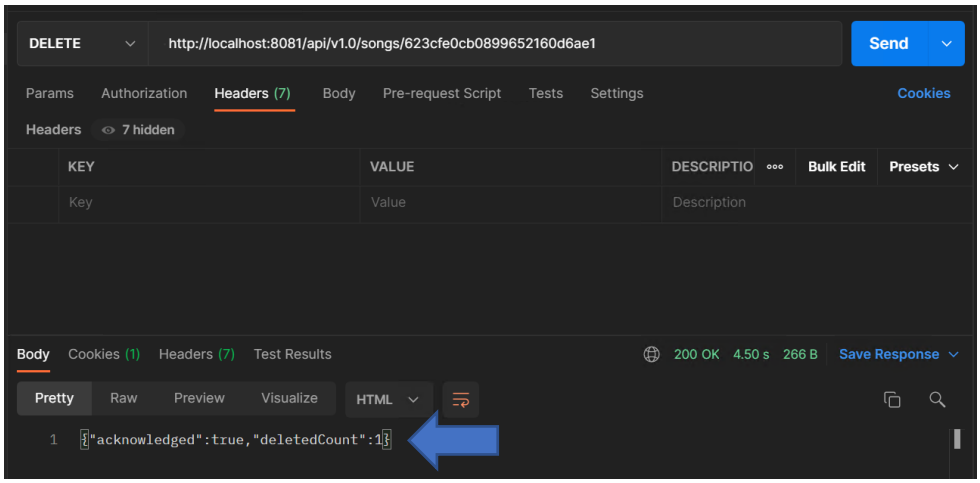
En primer lugar, creamos una **colección** pulsando en el botón **"+"** y luego la **petición** haciendo clic en el enlace ***"Add a request":***



A continuación, realizamos la petición DELETE a la API empleando la ***id de una canción existente en la base de datos***:

1. En el desplegable, seleccionamos el **método DELETE**.
    a. En la URL, escribimos: https://localhost:4000/api/v1.0/songs/:id
    b. Donde: **id es la id de una canción que esté almacenada en la base de datos**.
2. Hacemos clic en el **botón Send**.

Una vez enviada la petición, obtendremos la respuesta:



Observamos que estamos devolviendo directamente la respuesta de la base de datos en la que se incluyen los siguientes resultados **{"acknowledged": true, "deletedCount":1}** relativa al número de documentos afectados por el borrado. **Obviamente, sería recomendable devolver una respuesta más significativa**. Por ejemplo, si quisiéramos devolver los datos de la canción borrada podríamos modificar en el repositorio el método ***deleteOne()*** por ***FindOneAndDelete().*** Este último nos devuelve los datos del recurso borrado.

Crear un recurso canción - Método HTTP - POST

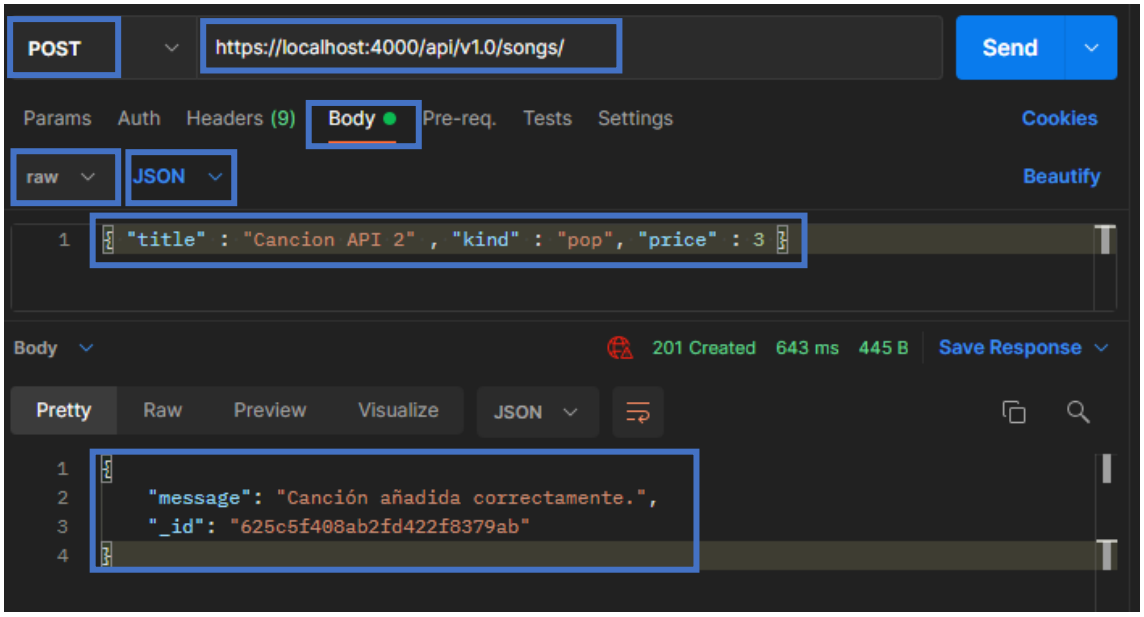
Para añadir un recurso canción utilizaremos una petición **POST** a **/api/v1.0/songs**. La respuesta a la petición incluirá el código de respuesta **201 – Created**, además de un mensaje en JSON. **Aunque no es posible que ocurra en este caso**, se incluye como ejemplo de aprendizaje el código de respuesta **409 – Conflict**. **Este código se utiliza cuando intentamos crear un recurso que ya existe en el sistema** o cuando intentamos actualizar (PUT) un recurso pasando la ID en la URL y en cuerpo, siendo éstas diferentes. En nuestro caso, el objeto JSON canción lo creamos sin especificar una _id, pues es MongoDB quien asigna ese identificador único.

```
app.post('/api/v1.0/songs', function (req, res) {
  try {
    let song = {
      title: req.body.title,
      kind: req.body.kind,
      price: req.body.price,
      author: req.session.user
    }
    // Validar aquí:  título, género, precio y autor.

    songsRepository.insertSong(song, function (songId) {
      if (songId === null) {
        res.status(409);
        res.json({error: "No se ha podido crear la canción. El recurso ya existe."});
      } else {
        res.status(201);
        res.json({
          message: "Canción añadida correctamente.",
          _id: songId
        })
      }
    });
  } catch (e) {
    res.status(500);
    res.json({error: "Se ha producido un error al intentar crear la canción: " + e})
  }
}) ;
```

Para asegurarnos de que funciona, probamos el servicio utilizando **Postman**. Para ello, debemos incluir un cuerpo (**Body**) a la petición en formato **raw y tipo JSON**, con el contenido:

```
{ "title" : "Cancion API 2" , "kind" : "pop", "price" : 3 }
```

Hacemos clic en el botón Send y como respuesta deberíamos obtener el mensaje ***"canción añadida correctamente",*** acompañado de la id de la canción, ver siguiente imagen:



Actualizar un recurso canción - Método HTTP PUT

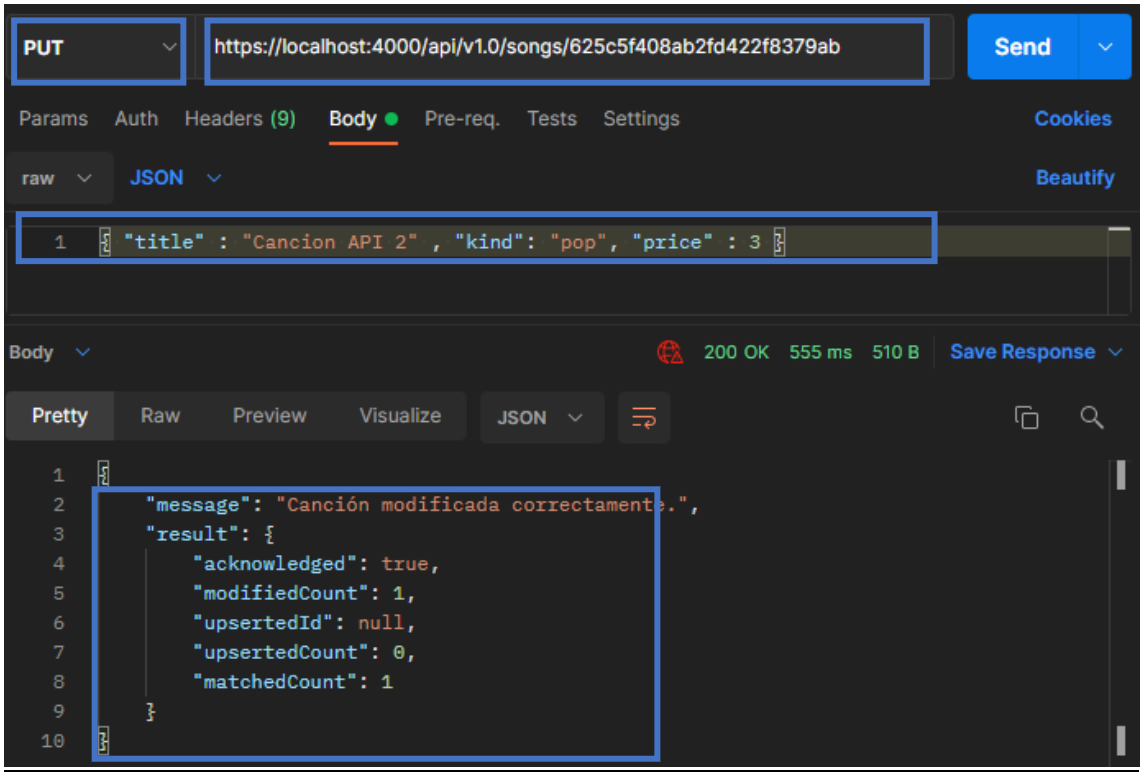
La función de actualización de una canción será muy similar a la de crear canción, pero **empleando el método PUT** (aunque también podríamos utilizar PATCH, para actualizar parcialmente un recurso). La petición PUT a **/api/v1.0/songs/:id** incluirá en el cuerpo los **datos que se deseen modificar** (el cliente podría querer modificar una o varias propiedades).

```
app.put('/api/v1.0/songs/:id', function (req, res) {
  try {
    let songId = ObjectId(req.params.id);
    let filter = {_id: songId};
    //Si la _id NO no existe, no crea un nuevo documento.
    const options = {upsert: false};
    let song = {
      author: req.session.user
    }
    if (typeof req.body.title !== "undefined" && req.body.title !== null)
      song.title = req.body.title;
    if (typeof req.body.kind !== "undefined" &&  req.body.kind !== null)
      song.kind = req.body.kind;
    if (typeof req.body.price !== "undefined" &&  req.body.price !== null)
      song.price = req.body.price;

    songsRepository.updateSong(song, filter, options).then(result => {
      if (result === null) {
        res.status(404);
        res.json({error: "ID inválido o no existe, no se ha actualizado la canción."});
      }
      //La _id No existe o los datos enviados no difieren de los ya almacenados.
      else if (result.modifiedCount == 0) {
        res.status(409);
        res.json({error: "No se ha modificado ninguna canción."});
```

```
        }
        else{
          res.status(200);
          res.json({
            message: "Canción modificada correctamente.",
            result: result
          })
        }
      }).catch(error => {
        res.status(500);
        res.json({error : "Se ha producido un error al modificar la canción."})
      });
    } catch (e) {
      res.status(500);
      res.json({error: "Se ha producido un error al intentar modificar la canción: "+ e})
    }
});
```

A continuación, probamos la actualización desde **Postman**, modificando parcialmente una de las canciones. Observamos que, **si enviamos los mismos datos que ya estaban almacenados, obtendremos una respuesta 409 – Conflict. El mismo caso ocurre si modificamos la _id por una que no exista y cumpla los requisitos de construcción de ids de MongoDB**.

Además, observamos que estamos devolviendo directamente la respuesta de la base de datos relativa al número de documentos afectados por la actualización (**result**). **Obviamente, sería recomendable devolver una respuesta más significativa**. Si quisiéramos devolver los datos de la canción actualizada podemos modificar en el repositorio el método ***updateOne()*** por ***FindOneAndUpdate(),*** este último nos devuelve los datos del recurso actualizado.



"SDI-IDGIT-10.1-SW API REST CRUD Canciones."

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

"SDI-2223-101-10.1-SW API REST CRUD Canciones."

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

Identificación del cliente con Token y control de acceso

Los **servicios web REST** no acostumbran a hacer uso del objeto sesión, por lo que la **identificación de los usuarios** se lleva a cabo comúnmente **utilizando un token de seguridad** que identifica al cliente que realiza la petición. Este **token acompaña a todas las peticiones y suele enviarse como parámetro GET, POST o en las HEADERS (esta última es la más común)**.

Aunque existen otros mecanismos de identificación basados en tokens la mayoría de APIs REST se basan en los siguientes:

1. **Token único:** cada cuenta de usuario es provista de un token único que la identifica y va asociado a todas las peticiones realizadas a servicios. Así, la aplicación controla: si el usuario tiene permisos para ejecutar los servicios, el número de veces que llama a los servicios, etcétera.
2. **Token por login:** cuando el cliente envía sus credenciales a un servicio específico recibe un token que debe ser almacenado y enviado en todas las peticiones que el cliente realice. Dependiendo de la implementación del servicio, este token puede funcionar de diferentes maneras, por ejemplo, caducando tras un tiempo sin recibir peticiones (algo similar a la sesión) o incluso caducando tras cada petición y reenviando uno nuevo.

Vamos a implementar la opción (2) **Token por login**. Crearemos un servicio para identificar al usuario asociado a la petición **POST** a **api/v1.0/users/login**, muy similar al login que implementamos en la aplicación web. A partir del email y el password encriptado del usuario, se realiza una búsqueda en la base de datos y, si hay coincidencia, devolvemos un JSON con el parámetro autenticado a true o false en caso contrario.

Incluir módulo jwt en app.js

Debemos modificar el sistema para que devuelva un **token de seguridad** cuando la autenticación sea correcta. Existen varias estrategias que podemos seguir para crear el token:

1. Crear un token totalmente aleatorio y almacenarlo junto a la información del usuario en la base de datos.
2. Crear un token con el identificador del usuario encriptado (y que nosotros podamos desencriptar para saber de qué usuario se trata). **Además del identificador del usuario el token puede contener otra información**, como la fecha de creación.
3. Otras muchas variaciones para incrementar el nivel de seguridad.

Optaremos por la (2). Encriptaremos el email del usuario + Timestamp, a través de Date.now() ya que incluir el tiempo actual es muy útil para implementar caducidades. Aunque podríamos utilizar el módulo crypto **para realizar estas encriptaciones**, vamos a optar por usar el **módulo**

jsonwebtoken https://www.npmjs.com/package/jsonwebtoken debido a su popularidad en este contexto. Descargamos el módulo accediendo desde la consola de comandos al directorio raíz del proyecto y ejecutando: **npm install jsonwebtoken**

```
PS C:\Dev\Projects\SDI_projects\sdiIDGIT-lab-nodejs\musicstoreapp> npm install jsonwebtoken

added 14 packages, and audited 117 packages in 3s

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

found 0 vulnerabilities
```

Dentro del fichero principal de la aplicación **app.js** declaramos el require del nuevo módulo y lo almacenaremos en la app, bajo la clave **jwt** (de esta forma podemos acceder a la variable jwt desde cualquier parte de la aplicación).

```
let app = express();
let jwt = require('jsonwebtoken');
app.set('jwt', jwt);
```

Endpoint para la autenticación de usuarios

Añadimos el siguiente endpoint en el módulo de la API REST:

```
app.post('/api/v1.0/users/login', function (req, res) {
  try {
    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.status(401); //Unauthorized
        res.json({
          message: "usuario no autorizado",
          authenticated: false
        })

      } else {
        let token = app.get('jwt').sign(
          {user: user.email, time: Date.now() / 1000},
          "secreto");
        res.status(200);
        res.json({
          message: "usuario autorizado",
          authenticated: true,
          token: token
        })
      }
    }).catch(error => {
      res.status(401);
      res.json({
        message: "Se ha producido un error al verificar credenciales",
        authenticated: false
      })
    })
  } catch (e) {
    res.status(500);
    res.json({
      message: "Se ha producido un error al verificar credenciales",
      authenticated: false
    })
  }
})
```

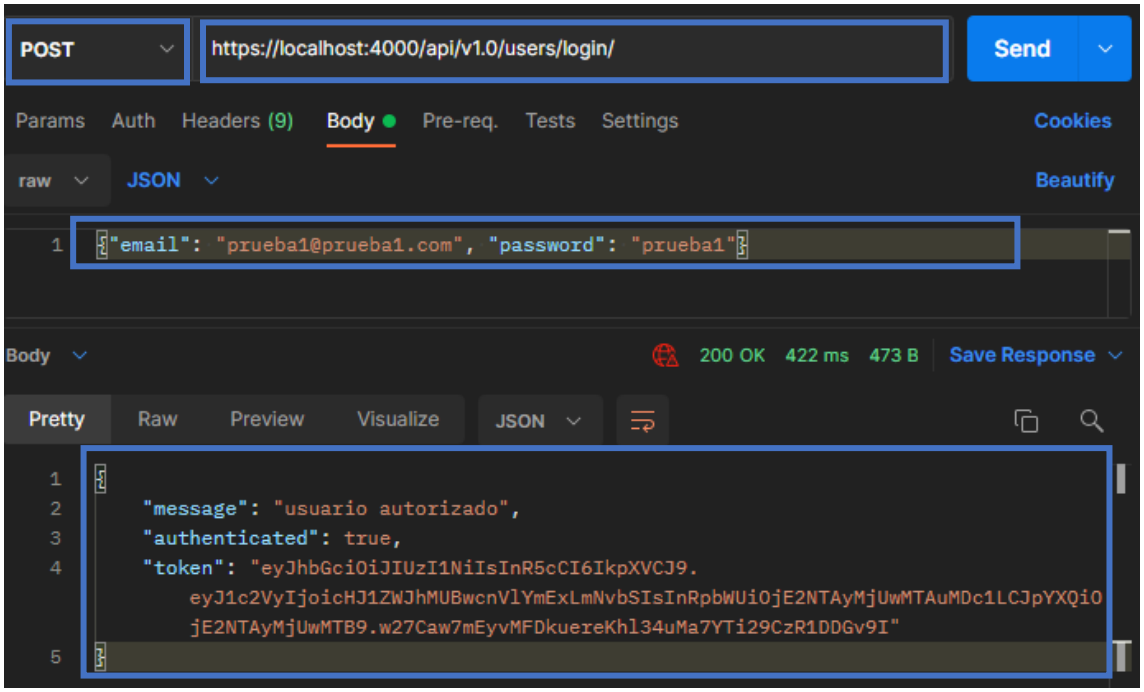
A continuación, incluimos como parametro el usersRepository en el módulo de la API REST (api/v1.0/songsAPIv1.0.js).

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

Finalmente, pasamos como parámetro el userRepository al el módulo de la API REST desde app.js.

```
require("./routes/users.js")(app, userRepository);
require("./routes/api/songsAPIv1.0.js")(app, songsRepository, userRepository);
```

Probamos a enviar una petición **POST** mediante Postman a **/api/v1.0/users/login**, pasando un email y una contraseña de un usuario que tengamos en la base de datos. Deberíamos obtener como **respuesta un código 200 (ok) y un JSON con el parámetro authenticated a true y un token**, ver la siguiente imagen:



Implementar un router de usuario - token

Vamos a requerir el ***uso de este token*** para cualquier endpoint de **/api/v1.0/songs/,** por lo que implementaremos un **nuevo Router en la carpeta routes (userTokenRoute).**

```
const jwt = require("jsonwebtoken");
const express = require('express');
const userTokenRouter = express.Router();
userTokenRouter.use(function (req, res, next) {
   console.log("userAuthorRouter");
   let token = req.headers['token'] || req.body.token || req.query.token;
   if (token != null) {
      // verificar el token
      jwt.verify(token, 'secreto', {}, function (err, infoToken) {
         if (err || (Date.now() / 1000 - infoToken.time) > 240) {
            res.status(403); // Forbidden
            res.json({
               authorized: false,
               error: 'Token inválido o caducado'
            });
         } else {
            // dejamos correr la petición
            res.user = infoToken.user;
            next();
         }
      });
   } else {
      res.status(403); // Forbidden
      res.json({
         authorized: false,
         error: 'No hay Token'
      });
   }
});
module.exports = userTokenRouter;
```

Obtenemos el parámetro token: admitimos que se envíe como parámetro: POST, GET o HEADER.

1. **Verificamos el token desencriptándolo**:
   a. **Si no conseguimos desencriptarlo o si han pasado más de 240 segundos** desde que se creó el token devolvemos un mensaje de **error**: "Token inválido o caducado."
   b. **Si lo desencriptamos dejamos correr la petición next().** Puede ser buena idea guardar el identificador del usuario en la respuesta: **res.user**. De esta forma no habrá que volver a desencriptar el token.
2. **Si no hay token, enviamos un mensaje de error** "No hay token".

Incluir nuevo router en app.js

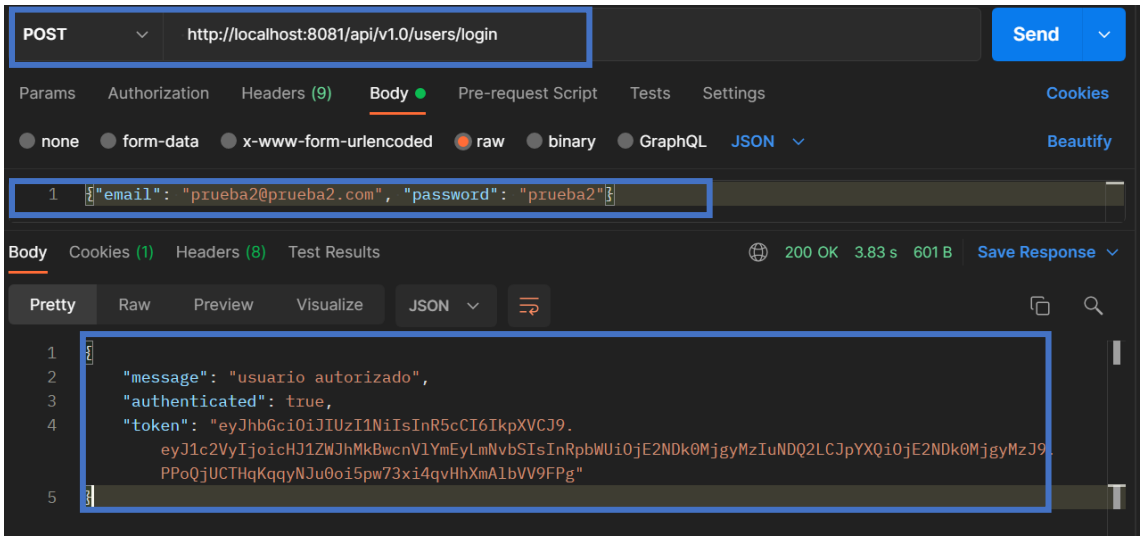
Incluimos el nuevo router en app.js para que solo usuarios con token autorizado puedan hacer peticiones a la API REST.

```
app.use("/songs/edit", userAuthorRouter);
app.use("/songs/delete", userAuthorRouter);

const userTokenRouter = require('./routes/userTokenRouter');
app.use("/api/v1.0/songs/", userTokenRouter);
...
```

Probar la aplicación

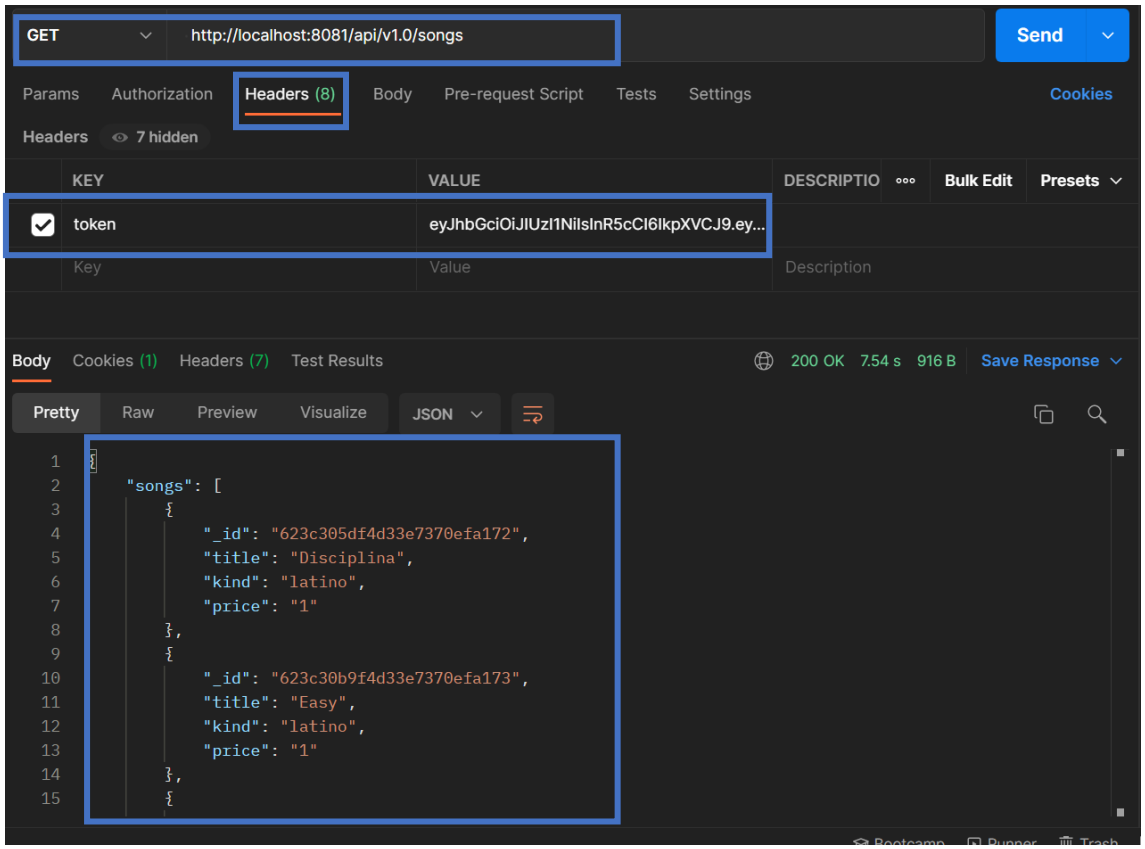
Guardamos los cambios, ejecutamos la aplicación e intentamos a acceder al servicio desde el navegador: http://localhost:8081/api/v1.0/songs/. Veremos que nos devolverá el siguiente mensaje:

Para probar en detalle todo lo implementado necesitaremos utilizar **Postman**. Primero hacemos una petición **POST** a http://localhost:8081/api/v1.0/users/login (Incluyendo en el body el email y password), **si la petición es correcta nos devolverá el token**.



A continuación, copiamos el token generado y creamos una nueva petición GET dirigida a https://localhost:8081/api/songs , **incluyendo una nueva cabecera Headers, con clave token y como valor el token obtenido**.

Al enviar la petición, deberíamos obtener el listado de canciones, tal y como muestra la siguiente imagen:



Además, habría que asegurarse que el usuario además de estar identificado también es el dueño de la canción cuando se trata de eliminar o modificar canciones. **Esta comprobación se puede añadir en las propias funciones afectadas (*recomendado) o creando un nuevo router**.

Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-10.2-SW API REST Autenticación."

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

"SDI-2223-101-10.2-SW API REST Autenticación."

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

Parte 2 - Cliente jQuery-Ajax

Access-Control-Allow-Origin

Por defecto y por razones de seguridad la aplicación ***tienda de música*** **no incluye las cabeceras Access-Controll-Allow-*** en sus respuestas, por lo que si implementamos un cliente JavaScript que haga **peticiones** contra esta aplicación, **nuestro navegador podría bloquearlas**.

Una solución durante la fase de desarrollo sería agregar en **app.js** las cabeceras más permisivas de Access-Cotrol-Allow-Origin para todas las peticiones. Añadimos el siguiente código en app.js:

```
let logger = require('morgan');
let app = express();

app.use(function(req, res, next) {
   res.header("Access-Control-Allow-Origin", "*");
   res.header("Access-Control-Allow-Credentials", "true");
   res.header("Access-Control-Allow-Methods", "POST, GET, DELETE, UPDATE, PUT");
   res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type,
Accept, token");
   // Debemos especificar todas las headers que se aceptan. Content-Type , token
   next();
});
```

Single Page Application (SPA)

A continuación, vamos a implementar una **aplicación que consuma los servicios web (API REST) de la tienda de música**. Será una **aplicación de una única página (SPA)** con el fin de proporcionar una **experiencia de usuario más fluida**, haciendo aparecer y desaparecer los componentes dinámicamente.

Existen muchas **tecnologías para desarrollar aplicaciones SPA**, en nuestro caso optaremos por la **combinación jQuery + AJAX**. Además, podemos usarla para **enriquecer interfaces de aplicaciones web** que no sean estrictamente SPA.

Cabe destacar que esta nueva aplicación podría ser desplegada dentro de una aplicación express o cualquier otro servidor web (mientras que admita HTML y JS). En este caso, para simplificar el desarrollo, vamos a hacer que esta aplicación sea parte de nuestra aplicación anterior **tienda de música**.

En la carpeta public/ del proyecto, creamos una subcarpeta ***apiclient*** y dentro un fichero ***client.html.*** Esta página será la **base de un cliente jQuery que utilizará los servicios web REST** de la tienda de música. Si teníamos más ficheros .html en el directorio **/public** los eliminamos.

Copiamos el siguiente contenido en el fichero client.html:

```
<html lang="en">
<head>
 <title>jQuery uoMusic </title>
 <meta charset="utf-8"/>
 <meta name="viewport" content="width=device-width, initial-scale=1"/>
 <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"/>
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></script>
 <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
</head>
<body>

<!-- Barra de Navegación superior -->
<nav class="navbar navbar-inverse">
 <div class="container-fluid">
  <div class="navbar-header">
   <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target="#myNavbar">
     <span class="icon-bar"></span>
     <span class="icon-bar"></span>
     <span class="icon-bar"></span>
   </button>
   <img src="/images/logo.png" alt="logo"/>
  </div>
  <div class="collapse navbar-collapse" id="myNavbar">
   <ul class="nav navbar-nav" id="barra-menu">
    <li><a onclick=widgetSongs()>Canciones</a></li>
   </ul>
   <ul class="nav navbar-nav navbar-right" id="barra-menu-derecha">
                    <!-- Opciones de menú aqui -->
   </ul>
  </div>
 </div>
</nav>

<!-- Contenido -->
<div class="container" id="main-container"> <!--  id para identificar -->
</div>

</body>
</html>
```

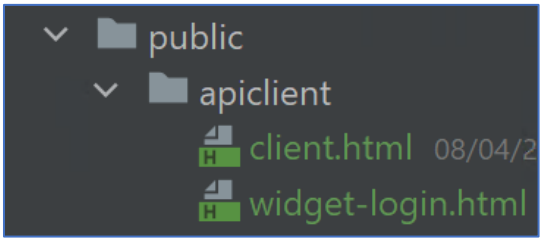
A través de client.html cargaremos dinámicamente diferentes ***componentes/widgets*** (un widget con el formulario de identificación, otro para mostrar las canciones, etc.) **que se mostrarán dentro del con id= "main-container"**.

Agregamos un script al final del fichero en el que incluiremos las variables globales que van a ser utilizadas en todo el cliente (**token** y **URLbase**) y aprovechamos también para cargar el **widget-login.html** sobre el contenedor principal (widget-login está aún sin implementar).

```
...
<!-- Contenido -->
<div class="container" id="main-container"> <!--  id para identificar -->
</div>
<script>
   let token;
   let URLbase = "http://localhost:8081/api/v1.0";
   $("#main-container").load("widget-login.html");
</script>
</body>
```

Sistema de autenticación (login)

Para definir un sistema de autenticación, creamos un nuevo fichero **widget-login.html** en la carpeta **/public/apiclient.**



Añadimos dos inputs, para el **email** y el **password**, y un **botón de envío.** En este caso el sistema de login va a ser ligeramente diferente al utilizado anteriormente:

1. No va a tener la etiqueta **<form>**, puesto que no se va a enviar por el método tradicional; **utilizaremos jQuery**.
2. Para facilitar el acceso a los elementos de la página HTML desde jQuery, es muy recomendable incluir **id** en todos los elementos que vayan a ser "manipulados".

```
<div id="widget-login">
 <div class="form-group">
  <label class="control-label col-sm-2" for="email">Email:</label>
  <div class="col-sm-10">
    <input type="email" class="form-control" name="email"
        placeholder="email@email.com" id="email" />
  </div>
 </div>
 <div class="form-group">
  <label class="control-label col-sm-2" for="password">Password:</label>
  <div class="col-sm-10">
    <input type="password"  class="form-control" name="password"
        placeholder="contraseña" id="password"/>
  </div>
 </div>
 <div class="form-group">
```

```
  <div class="col-sm-offset-2 col-sm-10">
   <button type="button" id="boton-login">Aceptar</button>
  </div>
 </div>
</div>
```

A continuación del código HTML implementaremos el script para registrar el clic en el **botón-login**. Como resultado, se envía el contenido de los campos **email** y **password** al servicio **POST en http://localhost:8081/api/v1.0/users/login.**

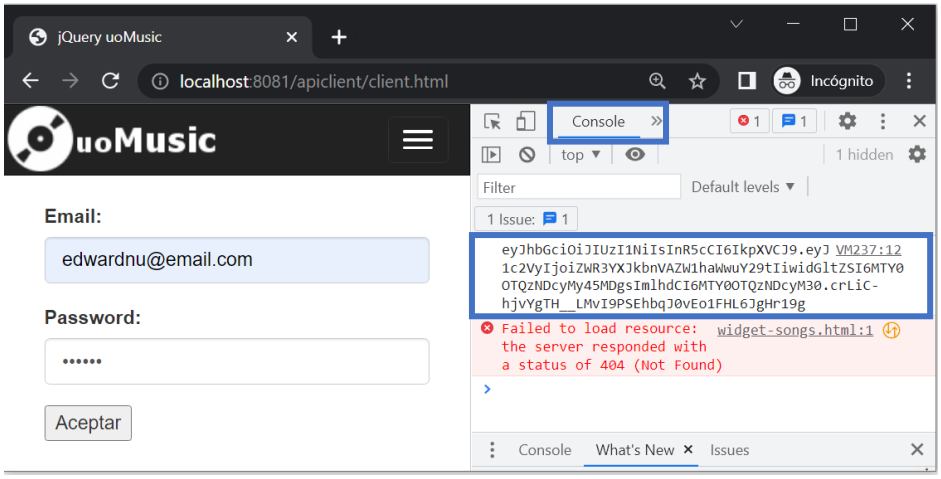
La petición se realiza a través del objeto $.ajax, esperando una respuesta en el formato especificado por **dataType** (JSON en nuestro caso). Concretamente, debería **devolvernos un objeto JSON con la propiedad token, cuyo valor almacenaremos en una variable global** para futuras peticiones.

Una vez realizada la autenticación con éxito**, vamos a mostrar al usuario automáticamente la lista de canciones**.

```
<script>
  $("#boton-login").click(function () {
    $.ajax({
      url: URLbase + "/users/login",
      type:"POST",
      data: {
        email: $("#email").val(),
        password: $("#password").val()
      },
      dataType: "json",
      success: function success(response) {
        console.log(response.token); // Prueba.
        token = response.token;
        $("#main-container").load("widget-songs.html");
      },
      error: function(error) {
        $("#widget-login")
        .prepend("<div class='alert alert-danger'>Usuario no encontrado</div>");
      }
    });
  });
</script>
```

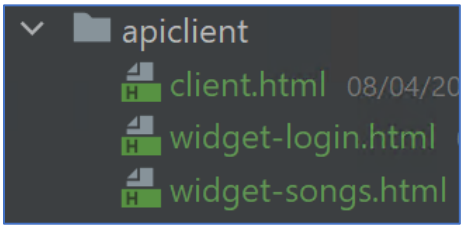
Es importante que el fragmento de <script> se añada después del HTML. En caso contrario, estamos intentando registrar un evento **click()** en un botón **botón-login** que aún no está creado. Otra opción para estar seguros de que los JS se ejecutan cuando toda la web está cargada es usar el *window.onload* o *$(document).ready*.

Accedemos a **http://localhost:8081/apiclient/client.html** y comprobamos que nos devuelve el **token** en la consola (tecla F12), ya que hemos incluido un mensaje **console.log(token)** una vez se identifica el usuario. El servicio debe estar activo para que el cliente funcione. Obviamente, **no pasará de la página de login ya que widget-songs.html aún no está implementado**.



Listar canciones

Al disponer del token de seguridad, ya es posible realizar una **petición al servicio web para obtener las canciones y mostrarlas** en nuestra aplicación. Comenzaremos **creando el fichero widget-songs.html:**



A continuación, definimos la **vista HTML** mediante una tabla con el cuerpo vacío. Es importante **identificar el cuerpo de la tabla con una id** ya que será donde **añadiremos las canciones**:

```
<div id="widget-songs" >
 <button class="btn" onclick="loadSongs()" >Actualizar</button>
 <table class="table table-hover">
  <thead>
  <tr>
   <th>Título</th>
   <th>Género</th>
   <th>Precio</th>
   <th class="col-md-1"></th>
  </tr>
  </thead>
  <tbody id="songsTableBody"></tbody>
 </table>
</div>
```

A continuación de la vista incluimos el script en el que definimos la función **loadSongs()**. Esta función hará una petición **GET** a **/api/v1.0/songs** para obtener las canciones e insertarlas en el cuerpo de la tabla (id=**songsTableBody**) a través de la función **updateSongsTable(songs)**.

Invocamos a la función principal **loadSongs ()** para que se invoque cuando se cargue la página.

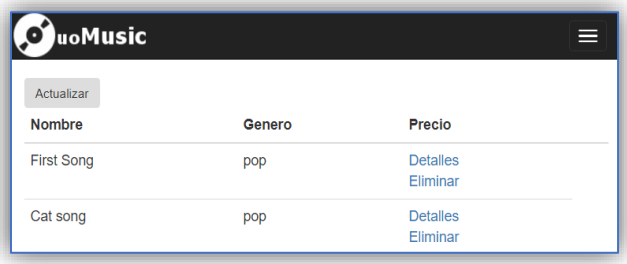
```
<script>
  var songs;

  function loadSongs() {
    $.ajax({
      url: URLbase + "/songs",
      type: "GET",
      data: {},
      dataType: 'json',
      headers: {"token": token},
      success: function (response) {
        songs = response.songs;
        updateSongsTable(songs);
      },
      error: function (error) {
        $("#main-container").load("widget-login.html");
      }
    });
  }

  function updateSongsTable(songs) {
    $("#songsTableBody").empty(); // Vaciar la tabla
    for (i = 0; i < songs.length; i++) {
      $("#songsTableBody").append(
        "<tr id=" + songs[i]._id + ">" +
        "<td>" + songs[i].title + "</td>" +
        "<td>" + songs[i].kind + "</td>" +
        "<td>" + songs[i].price + "</td>" +
        "<td>" +
        "<a onclick=songDetail('" + songs[i]._id + "')>Detalles</a><br>" +
        "<a onclick=songDelete('" + songs[i]._id + "')>Eliminar</a>" +
        "</td>" +
        "</tr>");
      // Mucho cuidado con las comillas del eliminarCancion
      //la id tiene que ir entre comillas ' '
    }
  }

  loadSongs();
</script>
```

En cada registro de la tabla hemos incluido dos botones con enlaces a las funciones **songDetail (id)** y **songDelete(id)**. Debemos de ser muy cuidadosos al hacer **appends** de código HTML que necesite comillas. En el caso del parámetro de las funciones **songDetail(id) y songDelete(id)**, si insertamos *onclick=(a3445a3)* en la página no funcionará; necesitamos *onclick=('a3445a3')*. *La id es un valor string, no el nombre de una variable.*

Comprobamos que la lista de canciones se muestra al identificarnos. El botón **Actualizar**, simplemente invoca de nuevo a la función **loadSongs** ().

Si quisiéramos que las canciones se actualizasen de forma automática cada N segundos (de forma similar a los clientes de correo electrónico) podríamos incluir un **setInternval**.

```
setInterval(function() {
   loadSongs();
}, 5000);
```



En el fichero **client.html implementamos la función widgetSongs(),** que se ejecutará al pulsar la opción del menú "***Canciones***". La utilizaremos **para permitir que el usuario vuelva a la lista de canciones**.

```
function widgetSongs(){
   $( "#main-container" ).load( "widget-songs.html");
}
```

Eliminar una canción

Implementaremos la función **songDelete(_id)** en **widget-songs.html.** Si la petición de eliminar se completa con éxito **(success)**, eliminamos el **<tr>** correspondiente a la canción. Recuerda incluir la función dentro de bloque <script>:

```
function songDelete(_id) {
   $.ajax({
      url: URLbase + "/songs/" + _id,
      type: "DELETE",
      data: {},
      dataType: 'json',
      headers: {"token": token},
      success: function (response) {
         console.log("Canción eliminada: " + _id);
         $("#" + _id).remove(); // eliminar el <tr> de la canción
      },
      error: function (error) {
         $("#main-container").load("widget-login.html");
      }
   });
}
```

Comprobamos que la acción eliminar funciona de forma correcta.

Ver detalle de una canción

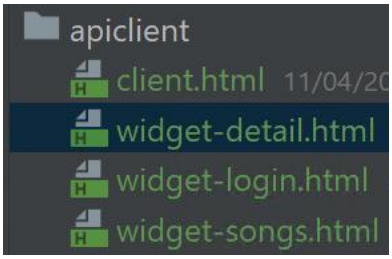
A continuación, implementaremos la función **songDetail(id) y la variable global selectedSongId** en el fichero widget-songs.html**.** Esta función se invoca al pulsar en el enlace *detalles* de cada canción y debe:

1. Guardar en la variable global **selectedSongId** la canción seleccionada. Esta variable va utilizarse para intercambiar datos entre widgets, anteriormente habíamos hecho algo similar con la variable **token** (declaramos **selectedSongId** al principio de widget-songs.html)
2. Cargar la vista **widget-detail.html**, la cual implementaremos posteriormente.

```
<script>
  var selectedSongId;
  var songs;
...
```

```
function songDetail(_id) {
  selectedSongId  = _id;
  $("#main-container").load("widget-detail.html");
}
```

Para mostrar los detalles crearemos el fichero widget-detail.html



Como en el caso anterior, incluimos primero la vista en HTML. Recordad: **es importante incluir ids** en los elementos que queramos editar.

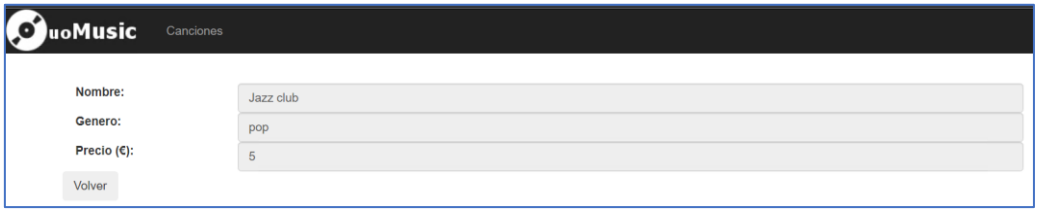
```
<div id="widget-detalles">
 <div class="form-group">
  <label class="control-label col-sm-2" for="title">Nombre:</label>
  <div class="col-sm-10">
   <input type="text" class="form-control" name="title"
        placeholder="Título de la canción"  id="title" readonly/>
  </div>
 </div>
 <div class="form-group">
  <label class="control-label col-sm-2" for="kind">Género:</label>
  <div class="col-sm-10">
   <input type="text" class="form-control" name="kind"
        placeholder="Género de la canción"  id="kind" readonly/>
```

```
  </div>
 </div>
 <div class="form-group">
  <label class="control-label col-sm-2" for="price">Precio (€):</label>
  <div class="col-sm-10">
   <input type="number" step="0.01" class="form-control" name="price"
        placeholder="2.50" id="price"  readonly/>
  </div>
 </div>
 <button onclick="widgetSongs()" class="btn" >Volver</button>
</div>
```

A continuación, vamos a incluir en **widget-detail.html** un script. Supondremos que la variable **selectedSongId** contiene el id de la canción que queremos cargar, realizaremos la petición para obtener la canción y después cargamos los valores en los inputs.

```
<script>
  $.ajax({
    url: URLbase + "/songs/" + selectedSongId,
    type: "GET",
    data: {},
    dataType: 'json',
    headers: {
      "token": token
    },
    success: function (response) {
      $("#title").val(response.song.title);
      $("#kind").val(response.song.kind);
      $("#price").val(response.song.price);
    },
    error: function (error) {
      $("#main-container").load("widget-login.html");
    }
  });
</script>
```

Comprobamos que la aplicación muestra los detalles de la canción correctamente. Al pulsar en el enlace detalle, se mostrará una vista como la siguiente:



Registrar una canción

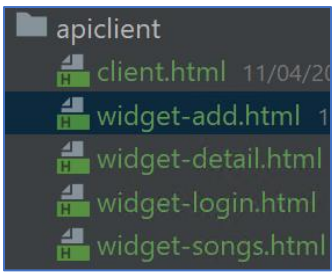
Para registra una nueva canción, primero abrimos el fichero **widget-songs.html** y añadimos un nuevo botón en su vista HTML.

```
<div id="widget-songs">
  <button class="btn btn-primary" onclick="widgetAddSong()">Nueva Canción</button>
  <button class="btn" onclick="loadSongs()">Actualizar</button>
...
```

Este botón va a ejecutar la función **widgetAddSong(),** la cual nos dirigirá al widget que muestra el formulario para registrar una nueva canción.

```
function widgetAddSong() {
  $( "#main-container" ).load( "widget-add.html");
}

loadSongs();
```

A continuación, creamos el fichero **widget-add.html** en la carpeta **/public/apiclient.**



Tanto los inputs a los que tenemos que acceder como el botón de agregar canción están identificados con una id (hay que recordar que las ids deben ser únicas, **no puede haber dos elementos con la misma id**). El botón invocará a la función **widgetAddSong ()**

```
<div id="widget-agregar" >
 <div class="form-group">
  <label class="control-label col-sm-2" for="title-add">Nombre:</label>
  <div class="col-sm-10">
   <input type="text" class="form-control" name="title-add"
        placeholder="Título de la canción"  id="title-add" />
  </div>
 </div>
 <div class="form-group">
  <label class="control-label col-sm-2" for="kind-add">Género:</label>
  <div class="col-sm-10">
   <input type="text" class="form-control" name="kind-add"
        placeholder="Género de la canción"  id="kind-add" />
  </div>
 </div>
 <div class="form-group">
```

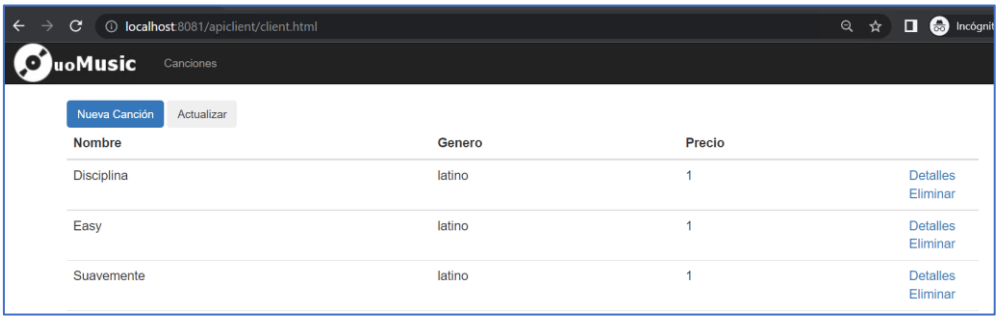
```
  <label class="control-label col-sm-2" for="price-add">Precio (€):</label>
  <div class="col-sm-10">
   <input type="number" step="0.01" class="form-control" name="price-add"
        placeholder="2.50" id="price-add"  />
  </div>
 </div>
 <div class="col-sm-offset-2 col-sm-10">
  <button type="button" class="btn btn-primary" id="boton-add"
       onclick="widgetAddSong()">Nueva canción</button>
 </div>
</div>
```

Finalmente, implementamos la función **widgetAddSong()** dentro de **widget-add.html** que tendrá la siguiente funcionalidad:

1. Obtiene los datos de la canción de los inputs declarados en la vista HTML utilizando sus ids y realiza una petición **POST** a **http://localhost:8081/api/v1.0/songs**.
2. Una vez registrada la canción, vuelve a mostrar la lista de canciones.

```
<script>
 function widgetAddSong( ) {
  $.ajax({
   url: URLbase + "/songs",
   type: "POST",
   data: {
    title: $("#title-add").val(),
    kind: $("#kind-add").val(),
    price: $("#price-add").val()
   },
   dataType: 'json',
   headers: {"token": token},
   success: function (response) {
    console.log(response); // <-- Prueba
    $("#main-container").load("widget-songs.html");

   },
   error: function (error) {
    $("#main-container").load("widget-login.html");
   }
  });
 }
</script>
```

Ejecutamos la aplicación y comprobamos que nos permite registrar nuevas canciones.



Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-10.3-SW Cliente JQuery + AJAX."

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

"SDI-2223-101-10.3-SW Cliente JQuery + AJAX."

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

Ampliación - Filtrado dinámico

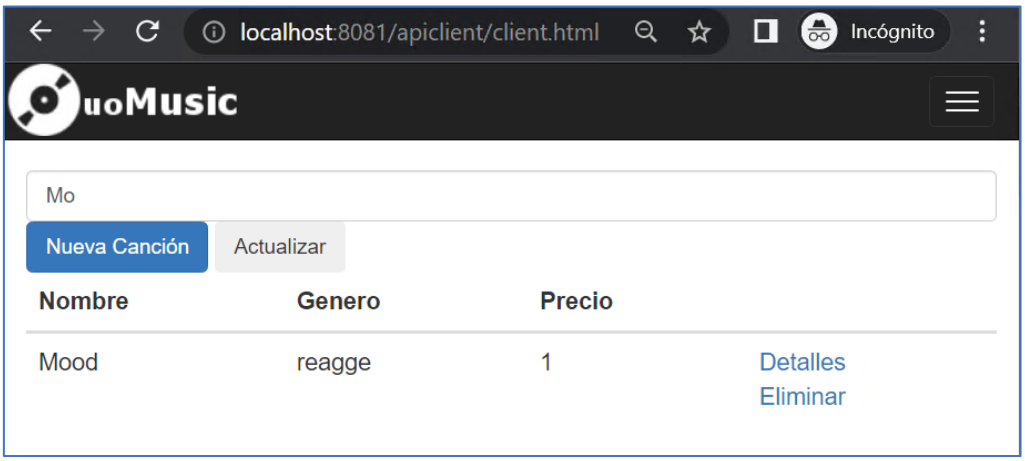
En este apartado, implementaremos un sistema de filtrado en la lista de canciones **widget-songs.html de forma que se muestren solo las canciones que coincidan con una cadena de texto.** Abrimos el fichero **widget-songs.html** e incluimos un input en la parte superior con la id filtro-nombre (para poder referenciarlo).

```
<div id="widget-songs">
  <input type="text" class="form-control" placeholder="Filtrar por nombre" id="filter-by-name"/>
  <button class="btn btn-primary" onclick="widgetAddSong()">Nueva Canción</button>
```

En la lógica **implementaremos un listener sobre el input con id filter-by-name** para que, cada vez que se detecte un cambio, recorra la lista de canciones originales y guarde en el array **filteredSongs** las canciones que tienen coincidencia con el texto introducido. Finalmente representamos las canciones presentes en el array **filteredSongs**. Añadimos el siguiente código **dentro de la etiqueta script de widget-songs.html**:

```
$('#filter-by-name').on('input',function(e){
  let filteredSongs = [];
  let filterValue = $("#filter-by-name").val();
  filteredSongs = songs.filter(song => song.title.toLowerCase().includes(filterValue.toLowerCase()));
  updateSongsTable(filteredSongs);
});
```

A continuación, comprobamos la funcionalidad:



Ampliación - Ordenación

En el propio **widget-songs.html** incluiremos un sistema de ordenación por título y precio. Incluimos en las cabeceras de la tabla (Título y Precio) los enlaces que invocarán respectivamente a **sortByTitle()** y **sortByPrice().**

```
<table class="table table-hover">
  <thead>
  <tr>
    <th>Título</th>
    <th><a onclick="sortByTitle()">Título</a></th>
    <th>Género</th>
    <th><a onclick="sortByPrice()">Precio</a></th>
    <th>Precio</th>
```

A continuación, implementamos las funciones **sortByTitle() y sortByPrice() que ordenarán el array utilizando la función sort**. Dependiendo del retorno de la función pasada, ordenará colocará una canción (a) delante o detrás otra (b):

- > 0 la canción **a** tiene más prioridad.
- < 0 la canción **b** tiene más prioridad.
- = 0 las canciones tienen la misma prioridad.

Por el momento ordenamos el precio de mayor a menor y los nombres de menor a mayor.

```
function sortByPrice() {
  songs.sort(function (a, b) {
    if (parseFloat(a.price) > parseFloat(b.price)) return -1;
    if (parseFloat(a.price) < parseFloat(b.price)) return 1;
    return 0;
  });
  updateSongsTable(songs);
}

function sortByTitle() {
```

```
  songs.sort(function (a, b) {
    if (a.title > b.title) return 1;
    if (a.title < b.title) return -1;
    return 0;
  });
  updateSongsTable(songs);
}
```

Si ejecutamos la aplicación y pulsamos sobre las cabeceras **Título** o **Precio** de la tabla modificaremos el orden por defecto.

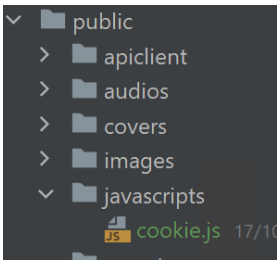
Si quisiésemos que **con cada clic sobre el enlace se invirtiera la ordenación**, bastaría con guardar una variable booleana con el estado de cada orden y, en función del estado, elegir un orden u otro para invertir el estado. Cambiamos la función sortByPrice() por el siguiente código:

```
var priceDesc = true;

function sortByPrice() {
  if (priceDesc) {
    songs.sort(function (a, b) {
      return parseFloat(a.price) - parseFloat(b.price);
    });
  } else {
    songs.sort(function (a, b) {
      return parseFloat(b.price) - parseFloat(a.price);
    });
  }
  updateSongsTable(songs);
  priceDesc = !priceDesc //invertir ordenación
}
```

Ampliación - Cookies

Podríamos almacenar el **token** en una cookie para que la información no se pierda al refrescar el cliente. Ahora mismo, cada vez que hacemos una actualización del navegador nos vuelve al login, lo cual es lógico, se reinicia todo el cliente.

JavaScript contiene un motor de cookies nativo, pero en este caso usaremos una librería https://github.com/js-cookie/js-cookie que simplifica notablemente su uso. Descargamos el fichero **cookie.js** del OneDrive y lo copiamos en la carpeta **/public/javascripts/.**



Añadimos la referencia al fichero **cookie.js** en el fichero principal **client.html.**

```
...
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
  <script src="/javascripts/cookie.js"></script>
</head>
```

Editamos el script de código del fichero **widget-login.html** para que, al recibir el token, se almacene en una cookie bajo la clave **token**. Al realizar un intento de autenticación fallido debemos eliminar la cookie **token**.

```
...
success:function success(response){
  console.log(response.token); // <- Prueba
  token = response.token;
  Cookies.set('token', response.token);
  $("#main-container").load("widget-songs.html");
},
error: function (error) {
  Cookies.remove('token');
  $("#widget-login")
    .prepend("<div class='alert alert-danger'>Usuario no encontrado</div>");
}
...
```

En el script del fichero **client.html** comprobaremos si hay una cookie **token** activa y, en caso afirmativo, la guardamos en la variable token (como si ya hubiésemos pasado por el login) e intentamos cargar el **widget-songs**.

```
<script>
  let token;
  let URLbase = "http://localhost:8081/api/v1.0";
  $("#main-container").load("widget-login.html");
```

```
  if ( Cookies.get('token') != null ){
     token = Cookies.get('token');
     $( "#main-container" ).load("widget-songs.html");
  }else {
     $( "#main-container" ).load("widget-login.html");
  }

...
```

A partir de este momento si nos hemos identificado y refrescamos la página no debería pedirnos identificarnos de nuevo.

Ampliación - Rutas

Como toda la aplicación se encuentra sobre la ruta **http://localhost:8081/apiclient/client.html**, esto **imposibilita compartir o referenciar URLs a partes concretas de la aplicación**. Aunque esto puede no ser un problema, en algunas aplicaciones podría ser interesante requerir esta funcionalidad.

Debemos implementar un **sistema de URLs empleando parámetros GET**. En este caso, vamos a manejar dos posibilidades: **client.html?w=login**, **client.html?w=songs**.

Al inicio del script de **widget-login.html** incluimos la **sentencia que modifica la URL del navegador**, añadiendo **?w=login**

```
<script>
   window.history.pushState("", "", "/apiclient/client.html?w=login");
   $("#boton-login").click(function () {
...
```

Repetimos el proceso en **widget-songs.html** , agregando **?w=songs**

```
<script>
   window.history.pushState("", "", "/apiclient/client.html?w=songs");
   var selectedSongId;
   var songs;
...
```

De esta forma, el parámetro w se va a agregar a la URL cuando el cliente cargue las vistas.

Para que el cliente (**client.html)** sea capaz de **cargar los componentes adecuados en función del parámetro w**, utilizaremos la función **searchParams**.

```
<script>
   let token;
   let URLbase = "http://localhost:8081/api/v1.0";

   if ( Cookies.get('token') != null ){
      token = Cookies.get('token');
      $( "#main-container" ).load("widget-songs.html");
      let url = new URL(window.location.href);
```

```
    let w = url.searchParams.get("w");

    if (w == "login") {
      $("#main-container").load("widget-login.html");
    }
    if (w == "songs") {
      $("#main-container").load("widget-songs.html");
    }

  }else {
    $( "#main-container" ).load("widget-login.html");
  }
...
```

Probamos el funcionamiento de las URLs:

- **http://localhost:8081/apiclient/client.html?w=login**
- **http://localhost:8081/apiclient/client.html?w=songs**

Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-10.4-SW Cliente JQuery + AJAX (Ampliaciones)."

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

"SDI-2223-101-10.4-SW Cliente JQuery + AJAX (Ampliaciones)."

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

Parte 3 – Consumir un servicio web REST desde Node.js

Cliente REST en Node.js

Vamos a **implementar un cliente REST dentro de la aplicación *tienda de música***. Este cliente va a utilizar el servicio web REST https://www.freeforexapi.com/ que permite hacer transformaciones de divisas.

Para realiza peticiones a un servicio REST vamos a utilizar **el módulo request**. Abrimos la consola de comandos, accedemos al directorio principal de la aplicación y ejecutamos el comando **npm install request**.

```
PS C:\Dev\Projects\SDI_projects\sdiIDGIT-lab-nodejs\musicstoreapp> npm install request
npm WARN            har-validator@5.1.5: this library is no longer supported
npm WARN            uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.random() in certain
 circumstances, which is known to be problematic.  See https://v8.dev/blog/math-random for details.
npm WARN            request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142

added 42 packages, and audited 159 packages in 2s

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

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

Incluimos el **require del módulo en el fichero principal de app.j**s, guardamos el módulo en una variable de la aplicación con clave 'rest'.

```
let app = express();

let rest = require('request');
app.set('rest', rest);
```

Para conocer **el valor de cambio entre EUR y USD** utilizamos el servicio **GET https://www.freeforexapi.com/api/live?pairs=EURUSD** que, según la documentación, nos devolverá un objeto JSON.

En el **siguiente ejemplo, que no emula un caso real** (habría formas mejores de hacerlo), vamos a incluir la posibilidad de mostrar los precios de una canción en dólares dentro de la vista de detalles de canción.

En el **controlador route/songs.js buscaremos y modificaremos la función que responde a GET /songs/:id** con el fin de ofrecer el precio también en dólares.

Para ello vamos a recuperar el módulo almacenado en **app.get('rest') y utilizarlo para obtener el cambio EUR -> USD en el servicio de freeforexapi.com**.

Debemos declarar un objeto de configuración especificando la información de la petición a realizar: **url, method, headers** (hemos incluido una cabecera a modo de ejemplo, pero realmente el servicio no la procesa).

El módulo rest necesita dos parámetros para realizar la petición:

- El objeto de configuración.
- La función de callback con parámetros (error, response, body).

Como casi todos los módulos que hemos visto hasta el momento este también funciona de **forma asíncrona**. Por lo tanto, debemos incluir el código que queremos que se ejecute posteriormente dentro de la función de callback que le enviamos como parámetro.

El **parámetro response** de la función de callback contiene la respuesta completa enviada por el servicio web. Utilizando response **podemos acceder a todos los elementos de la respuesta**. El parámetro body nos da acceso al cuerpo de la respuesta, en este caso al objeto JSON que nos devuelve freeforexapi. Aunque sepamos que la respuesta está en formato JSON debemos convertirla, ya que se procesa inicialmente como un String.

El objeto respuesta contiene los atributos code y rates. Dentro de rates, nos encontramos el atributo EURUSD con los atributos timestamp y rate (que es el que necesitamos)

```
{"rates":{"EURUSD":{"rate":1.19008,"timestamp":1618080728}},"code":200}
```

Vamos a obtener el cambio de divisas y multiplicar el precio actual por el cambio para obtener el **precio en dólares. Agregamos una nueva variable usd a la canción que se envía a la vista view/songs/songs.twig**.

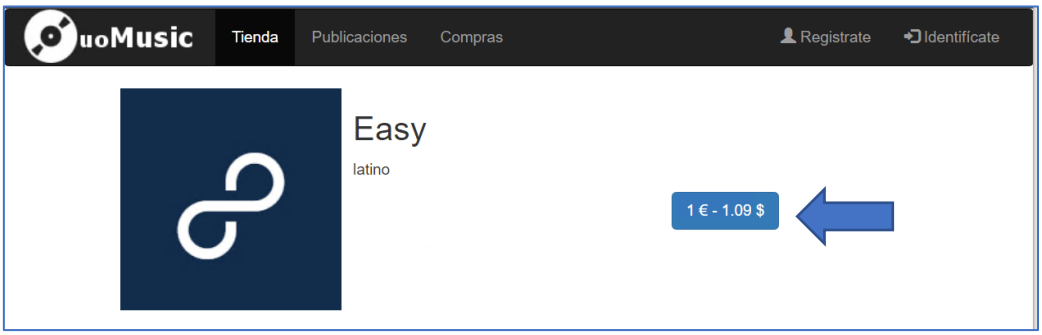
Nota: *Si hemos modificado la aplicación con algunos de los ejercicios complementarios* es posible que esta función sea necesario adaptarla al código de la versión que tengamos.

```
app.get('/songs/:id', function (req, res) {
  let songId = ObjectId(req.params.id)
  let filter = {_id: songId};
  let options = {};
  songsRepository.findSong(filter, options).then(song => {
    let comments_filter = {songId: songId};
    commentsRepository.getComments(comments_filter, options).then(comments => {
      let settings = {
        url: "https://www.freeforexapi.com/api/live?pairs=EURUSD",
        method: "get",
        headers: {
          "token": "ejemplo",
        }
      }
      let rest = app.get("rest");
      rest(settings, function (error, response, body) {
        console.log("cod: " + response.statusCode + " Cuerpo :" + body);
        let responseObject = JSON.parse(body);
        let rateUSD = responseObject.rates.EURUSD.rate;
        // nuevo campo "usd" redondeado a dos decimales
        let songValue= rateUSD * song.price;
        song.usd = Math.round(songValue * 100) / 100;
        res.render("songs/song.twig", {song: song, comments: comments});
      })
    }).catch(error => {
      res.send(" se ha producido un error")
    });
...
```

Finalmente mostramos el nuevo campo **song.usd** en la vista **views/songs/song.twig**. Para mostrarlo de forma rápida lo incluimos a continuación del precio original en el mismo enlace.

```
href="/songs/buy/{{ song._id }}">{{ song.price }} € - {{ song.usd }} $</a>
```

En la vista de detalles de las canciones los precios deberían verse de la siguiente forma:



Otros tipos de peticiones

En el ejemplo anterior solo hemos realizado una petición **GET,** pero el módulo request nos permite realizar todo tipo de peticiones, simplemente debemos cambiar el objeto configuración. **A continuación, se muestra un ejemplo (que no implementaremos) de una petición POST**. Este tipo de peticiones contienen un **body** en un formato especifico (JSON en el ejemplo).

```
// Insertar un anuncio
let ads = {
   description: 'Nuevo anuncio',
   price: '10'
};
let setting = {
   url: "http://ejemplo.ejemplo/anuncio",
   method: "POST",
   json: true,
   headers: {
      "content-type": "application/json",
   },
   body: add
}

let rest = app.get("rest");
rest(setting, function (error, response, body) {
...
})
```

Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-10.5-SW Consumir SW desde Node.js."

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

"SDI-2223-101-10.5-SW Consumir SW desde Node.js."

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

Parte 4 – Consumir un servicio web REST desde un cliente Java

En este apartado, crearemos un ***servicio web REST*** en Node.js y lo vamos a consumir desde un ***cliente en Java***.

Creando el Servicio web

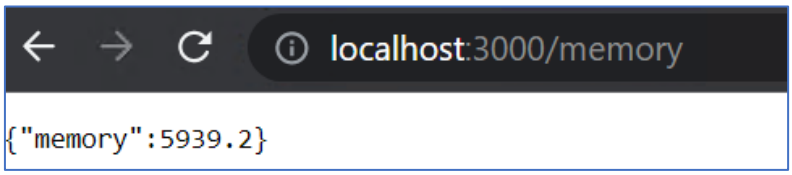
En la práctica 7 (primera práctica de Node.js), creamos un **fichero hello-world-server.js** cuyo contenido modificaremos para **crear un servicio web que responda a la petición GET /memory**. Este servicio nos devolverá un JSON con la memoria libre en el equipo. Si hemos borrado el fichero lo creamos de nuevo e incluimos el siguiente código:

```
let express = require('express');
let app = express();
let os = require('os');
let port = 3000;

app.get('/memory', function (req, res) {
  setTimeout(function () { // Espera de 10 segundos

    console.log(os.freemem());
    var freeMemory = os.freemem() / 1000000;  //pasar a MB
    res.status(200);
    res.json({
      memory: freeMemory
    });

  }, 10000);
});

app.listen(port, function () {
  console.log("Servidor listo " + port);
});
```

Para obtener la memoria de nuestro equipo usamos el módulo **os**. No hace falta descargarlo ya que se trata de un módulo nativo que nos da a acceso a varias funciones del sistema. **https://nodejs.org/api/os.html**

Importante: en lugar de responder automáticamente posponemos la respuesta 10 segundos utilizando para ello un **setTimeout**. Vamos a utilizar esta pausa para emular un servicio computacionalmente costoso.

Ejecutamos el fichero hellow-world-server.js (click derecho Run…) y accedemos a http://localhost:3000/memory. Deberíamos obtener una respuesta una vez hayamos esperado unos 10 segundos.



Nota: *Si falla la aplicación es posible que tengamos que incluir el modulo express en el fichero package.json e instalar express en la raíz del proyecto (NO en musicstoreapp). Si se ha borrado tambien el package.json entonces hay que crear el fichero con el siguiente contenido y ejecutar el comando npm install en la carpeta raíz del proyecto:*

Contenido del fichero package.json:

```
{
 "name": "memoryapp",
 "version": "0.0.0",
 "private": true,
 "scripts": {},
 "dependencies": {
  "express": "^4.18.2",
  "http-errors": "~1.6.3"
 }
}
```

Nota: Incluir el siguiente Commit Message ->

"SDI-IDGIT-10.6-SW Servicio Memoria."

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

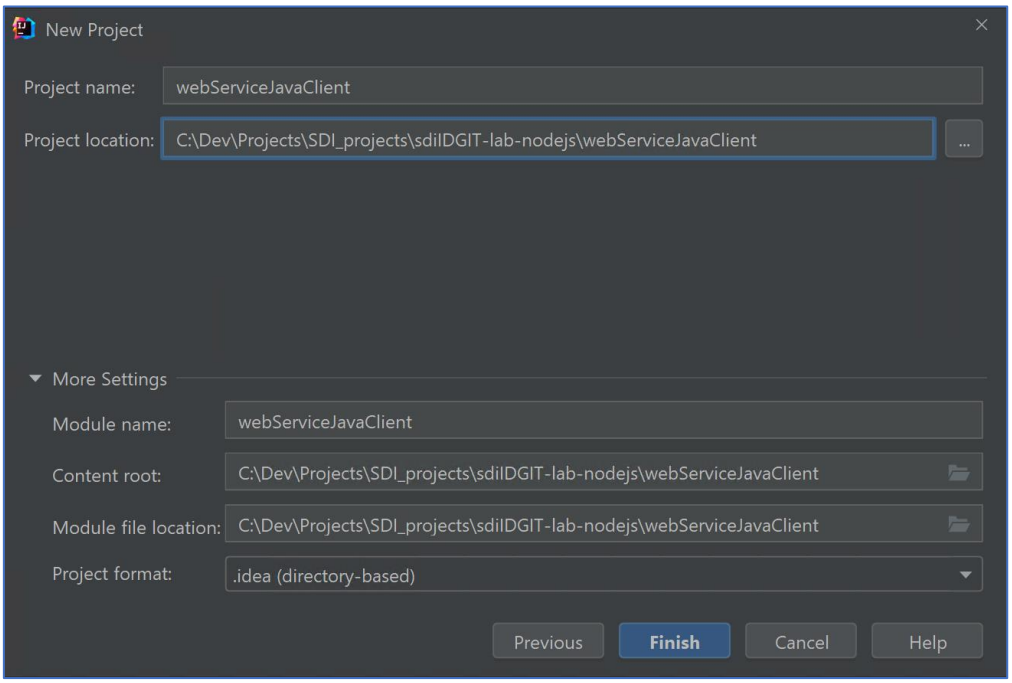
"SDI-2223-101-10.6-SW Servicio Memoria."

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

Creando el cliente Java

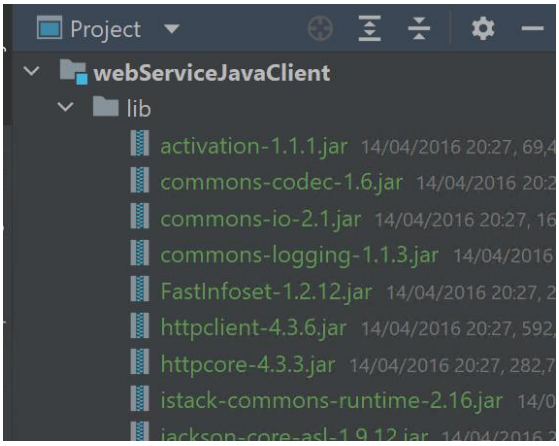
Vamos a desarrollar un proyecto de Java que nos permita consumir el servicio web creado en el aparatado anterior. Desde Intellij IDEA vamos a: **File | New Project | Java**.

El proyecto Java lo vamos a crear en una subcarpeta dentro del repositorio de Node.js (sdiIDGIT-lab-nodejs), cuyo nombre será: **webServiceJavaClient**:

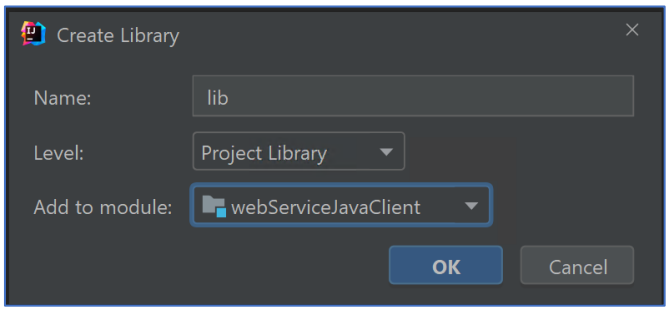


El cliente va a obtener el consumo de memoria de un equipo y lo mostrará en una ventana. Para la parte gráfica de la aplicación utilizaremos **Swing** y para realizar las llamadas al servicio web nos basaremos en la librería **JAX-RS** (existen otras muchas librerías para invocar servicios web REST desde Java).

Creamos una nueva carpeta en el proyecto y la llamamos **lib**, copiamos en el directorio las librerías a descargar de OneDrive (**lib.zip**). Entre ellas se encuentra **JAX-RS** y otras utilidades.



A continuación, añadimos al **build path** del proyecto todas las librerías de la carpeta **/lib,** pulsando el botón derecho sobre la carpeta lib **-> Add as Library.** Finalmente pulsamos en el botón **Ok**.



Implementando la clase Window

En la carpeta src definimos el paquete **com.uniovi.sdi** y, dentro del mismo, una nueva clase Java llamada **Window.**

Vamos a implementar una ventana con **swing** en la que definiremos:

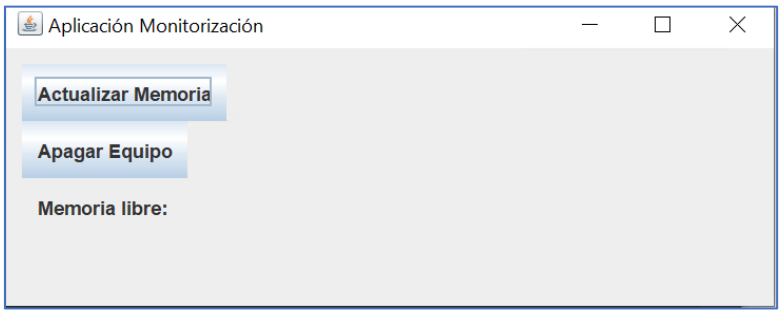
- Un Frame y un Panel.
- Un botón actualizar, que posteriormente realizará la llamada al servicio Web para consultar el consumo de memoria actual.
- Un botón apagar, que muestra un mensaje por pantalla (no tendrá funcionalidad real, es solo para comprobar si el interfaz de la aplicación responde).
- Un Label, en el que se mostrará la memoria libre actual.
- Una función **main** en la cual creamos una instancia de la ventana (Window).

Analizamos y copiamos el siguiente código en la clase Window:

```
package com.uniovi.sdi;

import javax.swing.*;
import javax.swing.border.EmptyBorder;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Window {
   public JLabel textMemory;
   JFrame frame;
   JPanel panel;
   JButton updateButton;
   JButton turnOffButton;
   int requests = 0;

   public Window() {
      // Frame
      frame = new JFrame("Aplicación Monitorización");
      frame.setSize(500, 200);
      frame.setLocationRelativeTo(null);

      // Panel
      panel = new JPanel();
      panel.setBorder(new EmptyBorder(10, 10, 10, 10));
      panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
```

```
        frame.add(panel);

        // Botón Actualizar
        updateButton = new JButton("Actualizar Memoria");
        updateButton.setBorder(new EmptyBorder(10, 10, 10, 10));
        updateButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent arg0) {

            }
        });
        panel.add(updateButton);

        // Botón Actualizar
        turnOffButton = new JButton("Apagar Equipo");
        turnOffButton.setBorder(new EmptyBorder(10, 10, 10, 10));
        turnOffButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent arg0) {
                JOptionPane.showMessageDialog(frame, "Enviado apagar!");
            }
        });
        panel.add(turnOffButton);
        // Texto memoria
        textMemory = new JLabel();
        textMemory.setBorder(new EmptyBorder(10, 10, 10, 10));
        textMemory.setText("Memoria libre:");
        panel.add(textMemory);

        // Propiedades visibilidad frame
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public static void main(String[] args) throws InterruptedException {
        new Window();
    }
}
```

Si ejecutamos la aplicación deberíamos ver la siguiente ventana:



Ahora nos falta implementar la lógica que se ejecuta al pulsar el **botón actualizar (updateButton)**. En esta funcionalidad deberíamos:

- Hacer una invocación al servicio REST **GET / http://localhost:3000/memory.**
    - Desde Intellij vamos al menú **File|Open Recent** y abrimos el proyecto Node.js en una nueva instancia de Intellij IDEA (opción **New Window**) y arrancamos el hello-world-server.js.
- Procesar el objeto JSON recibido {"memoria": 6740.967424}
- Mostrar la memoria consumida en el componente **textMemory**

La API JAX-RX está pensada para consumir servicios REST gestionando las peticiones HTTP de forma relativamente directa. Para realizar una petición debemos:

- Crear un objeto cliente (ClientBuilder.newClient())
- Especificar la URL a la que accedemos (.target(…))
- A partir de la URL, crear una petición (.request())
- **Añadir los tipos MIME en los que deseamos recibir** el contenido (.accept(…))
- Ejecutar el tipo de petición que necesitemos: GET, PUT, POST, DELETE (.get())
- Recoger los datos en el formato Java que necesitemos (.readEntity(…))

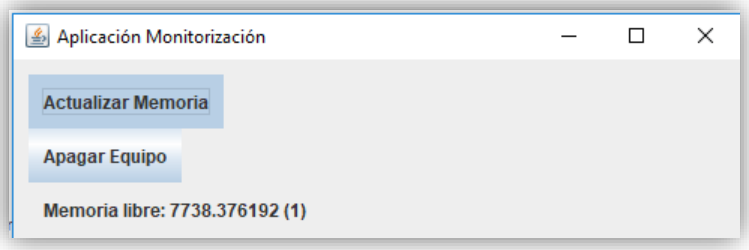
Completamos los parámetros para la petición a nuestro servicio. En el caso del parámetro especificado en la función **readEntity**() utilizaremos un tipo de objeto **ObjectNode**.

ObjectNode es un objeto genérico de la librería **codehaus** que nos sirve para almacenar objetos JSON de cualquier tipo. También podríamos haber creado una nueva clase **ResponseMemory** con un atributo **memory (get/set),** la cual hubiese servido para que se parseara automáticamente el JSON a un objeto de ese tipo. Añadimos el siguiente código en la clase Window:

```
...
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);

updateButton.addActionListener(arg0 -> {
    requests++;
    ObjectNode responseJSON;
    responseJSON = ClientBuilder.newClient()
            .target("http://localhost:3000/memory")
            .request()
            .accept(MediaType.APPLICATION_JSON)
            .get()
            .readEntity(ObjectNode.class);

    String memory = responseJSON.get("memory").toString();
    textMemory.setText("Memoria libre: " + memory + " (" + requests + ")");
});
}
```

Añadimos los imports necesarios, ejecutamos la aplicación y comprobamos lo que sucede al pulsar el botón **Actualizar Memoria**.

Durante los 10 segundos que tarda en completarse la petición, la aplicación está bloqueada. Lo sabemos porque no podemos pulsar el botón **Apagar Equipo.**



En la aplicación anterior, hemos lanzado la petición al servicio desde el hilo principal. Este hilo es el que se encarga de actualizar la interfaz gráfica de la aplicación. Mientras está esperando por la respuesta, toda la interfaz se mantiene bloqueada.

Nunca deberíamos realizar llamadas a servicios web ni operaciones de lógica de negocio costosas desde el hilo principal.

Petición desde hilo

Modificaremos la implementación anterior para que la petición se realice desde un nuevo hilo. Creamos la nueva clase **UpdateMemoryThread**. En esta clase implementamos un hilo (clase que hereda de Thread).

Creamos un constructor donde reciba una referencia a **Window** y una función **run()** donde se haga la misma invocación al servicio que antes realizábamos desde el botón.

Una vez obtenido el consumo de memoria, vamos a invocar al método **UpdateMemory(memory)** del objeto **Window**. **Este método está aún sin implementar, pero debería actualizar la memoria que se muestra por pantalla**. Incluimos el siguiente código en la clase UpdateMemoryThread:

```
package com.uniovi.sdi;

import org.codehaus.jackson.node.ObjectNode;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.MediaType;

public class UpdateMemoryThread extends Thread  {
  Window window;

  public UpdateMemoryThread(Window window) {
    this.window = window;
  }

  public void run() {
    ObjectNode responseJSON;
    responseJSON = ClientBuilder.newClient()
        .target("http://localhost:3000/memory")
```

```
            .request()
            .accept(MediaType.APPLICATION_JSON)
            .get()
            .readEntity(ObjectNode.class);
      String memory = responseJSON.get("memory").toString();
      window.updateMemory(memory);
    }}
```

Volvemos a la clase **Window**. Sustituimos el código que se ejecuta al pulsar el **updateButton**. Crearemos una instancia del hilo **UpdateMemoryThread** y lo lanzaremos (método **start()**)

```
updateButton.addActionListener(arg0 -> {
    requests++;
    UpdateMemoryThread thread = new UpdateMemoryThread(Window.this);
    thread.start();
/*  ObjectNode responseJSON;
    responseJSON = ClientBuilder.newClient()
            .target("http://localhost:3000/memory")
            .request()
            .accept(MediaType.APPLICATION_JSON)
            .get()
            .readEntity(ObjectNode.class);

    String memory = responsnseJSON.get("memory").toString();
    textMemory.setText("Memoria libre: " + memory + " (" + requests + ")");*/
});
```

Solo nos falta implementar el método **updateMemory(memory)**, en el que se actualiza la etiqueta de texto. La implementación más sencilla podría ser la siguiente:

```
public void updateMemory(String memory) {
    textMemory.setText("Memoria libre: " + memory + " (" + requests + ")");
}
```

Aunque este código puede que en ocasiones funcione correctamente, si lo utilizásemos, estaríamos cometiendo un error.

updateMemory() va a ser ejecutado desde un hilo secundario, no desde el hilo principal que recordamos que es el encargado de actualizar la Interfaz de usuario. **Si dos hilos modificaran simultáneamente el mismo elemento de interfaz se produciría una excepción**. Muchas tecnologías ni siquiera permiten que hilos distintos al principal modifiquen la interfaz de usuario.

Para conseguir que ese fragmento de código sea ejecutado por el hilo principal utilizamos el **SwingUtilities.invokeLater (Runnable).**

```
public void updateMemory(String memory) {
    SwingUtilities.invokeLater(() -> textMemory.setText("Memoria libre: " + memory + " (" +
requests + ")"));}
```

Ejecutamos la aplicación y debería funcionar sin bloqueos de interfaz, permitiéndonos pulsar los dos botones.

Añadimos a Git el nuevo proyecto: clic derecho en la carpeta de nuestro cliente Java (webServiceClientejava) → Git |Add y luego hacemos commit and push.

Nota: Incluir el siguiente Commit Message ->

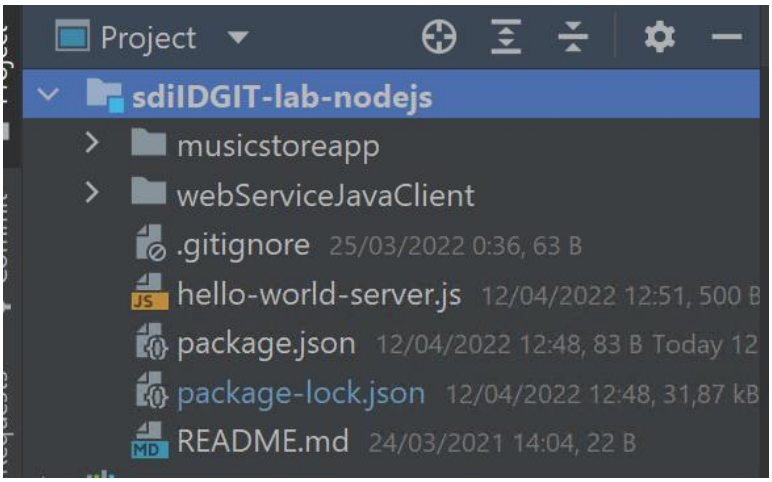
"SDI-IDGIT-10.7-SW Cliente Java."

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

"SDI-2223-101-10.7-SW Cliente Java."

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

En el repositorio de GitHub y en el entorno de Intellij IDEA deberíamos tener los siguientes proyectos (musicstoreapp, webServiceJavaClient y hello-world-server.js):



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-2223-101-10.1-SW API REST CRUD Canciones.**
- ⇒ **SDI-2223-101-10.2-SW API REST Autenticación.**
- ⇒ **SDI-2223-101-10.3-SW Cliente JQuery + AJAX.**
- ⇒ **SDI-2223-101-10.4-SW Cliente JQuery + AJAX (Ampliaciones).**
- ⇒ **SDI-2223-101-10.5-SW Consumir SW desde Node.js.**
- ⇒ **SDI-2223-101-10.6-SW Servicio Memoria.**