Sistemas Distribuidos e Internet
Desarrollo de aplicaciones Web con NodeJS
Sesión - 10
Curso 2019/ 2020

Contenido

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_musica**. 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** anteriormente implementados.
3. **Consumir un servicio web externo desde nuestra aplicación tienda_musica**.
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: https://es.wikipedia.org/wiki/Anexo:C%C3%B3digos_de_estado_HTTP.
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.stringfly** para transformarlos datos a objetos a JSON. Con **res.json** también podemos transformar objetos a formato JSON.

Gestión del recurso canción

En **la carpeta routes, creamos un nuevo fichero rapicanciones.js** en el que implementaremos una api REST.

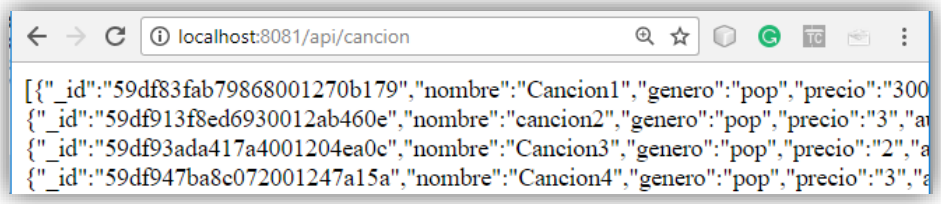
A través de una petición **GET** a **/api/cancion/** obtendremos una lista con todas las canciones en formato JSON.

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

    app.get("/api/cancion", function(req, res) {
        gestorBD.obtenerCanciones( {} , function(canciones) {
            if (canciones == null) {
                res.status(500);
                res.json({
                    error : "se ha producido un error"
                })
            } else {
                res.status(200);
                res.send( JSON.stringify(canciones) );
            }
        });
    });
}
```

Incluimos el nuevo controlador en **app.js**

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

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. A continuación, probaremos el servicio realizando la petición GET a través del navegador: https://localhost:8081/api/cancion .



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. "autor": prueba@prueba2.com
2. "autor": /usuario/prueba2@prueba2.com

La petición **GET** a **/api/cancion/:id** debería retornar la canción con el id correspondiente.

```
app.get("/api/cancion/:id", function(req, res) {
    var criterio = { "_id" : gestorBD.mongo.ObjectID(req.params.id)}

    gestorBD.obtenerCanciones(criterio,function(canciones){
        if ( canciones == null ){
            res.status(500);
            res.json({
                error : "se ha producido un error"
            })
        } else {
            res.status(200);
            res.send( JSON.stringify(canciones[0]) );
        }
    });
});
```

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

```
app.delete("/api/cancion/:id", function(req, res) {
    var criterio = { "_id" : gestorBD.mongo.ObjectID(req.params.id)}

    gestorBD.eliminarCancion(criterio,function(canciones){
        if ( canciones == null ){
            res.status(500);
            res.json({
                error : "se ha producido un error"
            })
        } else {
            res.status(200);
            res.send( JSON.stringify(canciones) );
        }
    });
});
```

Para **probar peticiones tipo: DELETE, POST, PUT;** podemos utilizar la aplicación **Postman.**

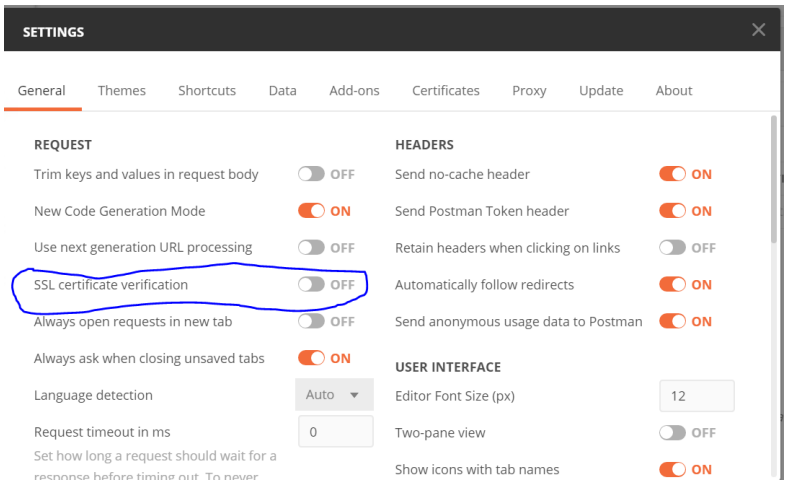
Para usar POSTMAN hay dos alternativas:

1) Instalar la aplicación nativa. https://www.postman.com/
2) Instalar la extensión de Chrome.
    a. https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop.
    b. Una vez instalada, accedemos a la misma a través de chrome://apps.

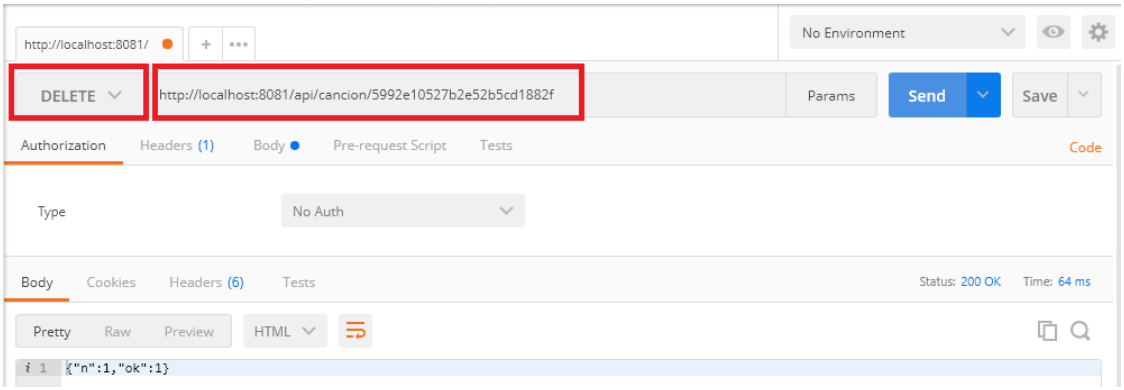
Nota 1: Si tienes problemas a ejecutar una petición por https cuando usas la extensión, tienes importar los certificados en Chrome. https://blog.postman.com/2019/07/17/self-signed-ssl-certificate-troubleshooting/

Nota 2: Si usas la aplicación de escritorio solo tienes que desactivar el botón de verificación de SSL en la configuración (Setting) de Postman.

Nota 3: En esta práctica usamos la aplicación nativa.



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



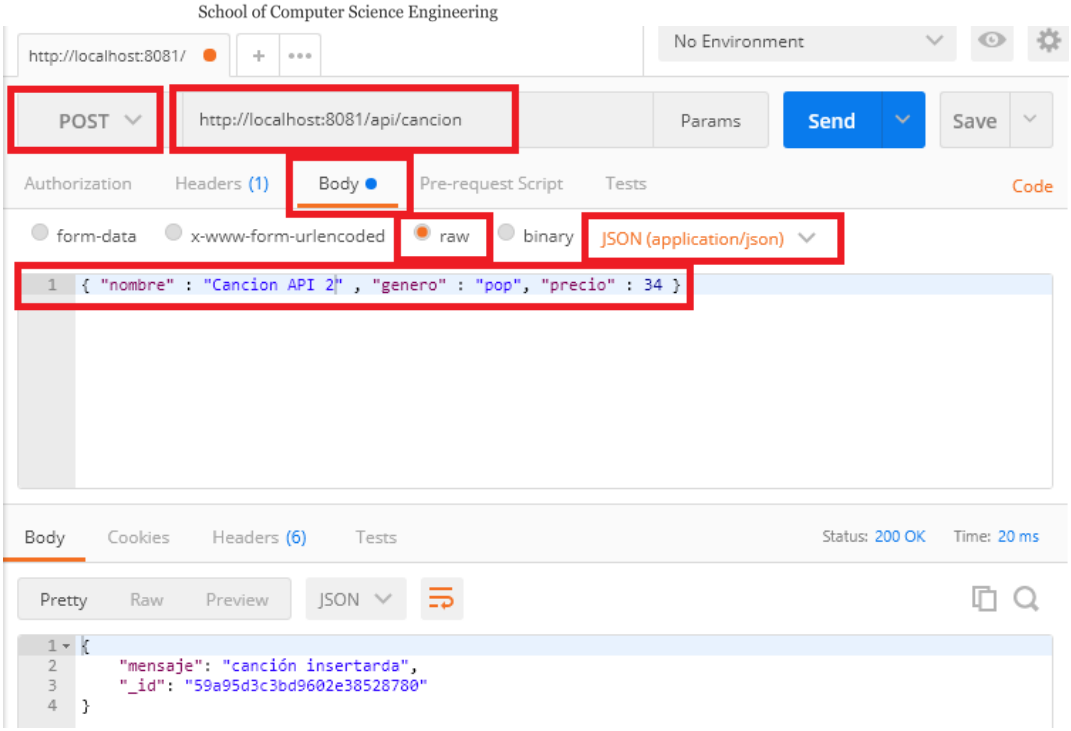
Observamos que estamos devolviendo directamente la respuesta de la base de datos en la que se incluyen los siguientes resultados ***{"n":1, …, "ok":1 …}*** relativa al número de documentos(n:1) afectados por el borrado. Obviamente, sería recomendable devolver una respuesta más significativa.

Para agregar un recurso canción utilizaremos una petición **POST** a **/cancion**. La respuesta a la petición incluirá el código de respuesta **201 – Created**, además de un mensaje en JSON.

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

    // ¿Validar nombre, genero, precio?

    gestorBD.insertarCancion(cancion, function(id){
        if (id == null) {
            res.status(500);
            res.json({
                error : "se ha producido un error"
            })
        } else {
            res.status(201);
            res.json({
                mensaje : "canción insertarda",
                _id : id
            })
        }
    });

});
```

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

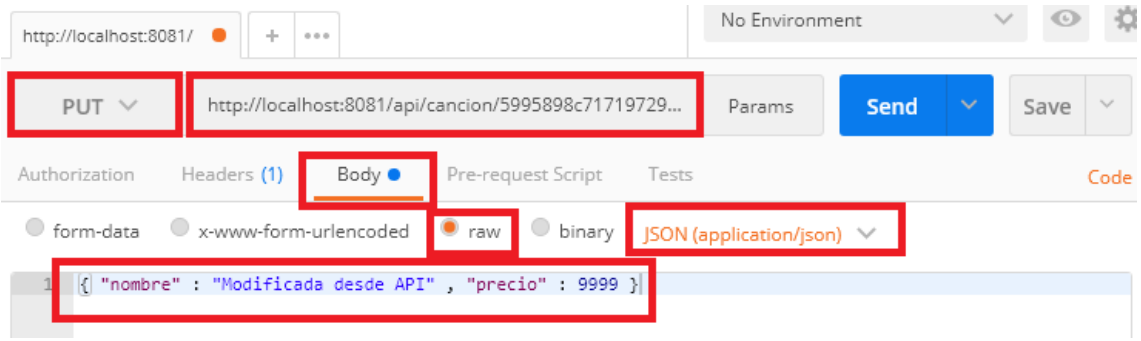
{ "nombre" : "Cancion API 2" , "genero" : "pop", "precio" : 34 }



La función de actualización de una canción será muy similar a la de crear canción pero **empleando el método PUT** (aunque también podríamos utilizar UPDATE). La petición PUT a **/api/cancion/:id** incluirá en el cuerpo los **datos que se deseen modificar** (el cliente podría querer modificar solo una propiedad o varias).

```
app.put("/api/cancion/:id", function(req, res) {

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

    let cancion = {}; // Solo los atributos a modificar
    if ( req.body.nombre != null)
        cancion.nombre = req.body.nombre;
    if ( req.body.genero != null)
        cancion.genero = req.body.genero;
    if ( req.body.precio != null)
        cancion.precio = req.body.precio;

    gestorBD.modificarCancion(criterio, cancion, function(result) {
        if (result == null) {
            res.status(500);
            res.json({
                error : "se ha producido un error"
            })
        } else {
            res.status(200);
            res.json({
                mensaje : "canción modificada",
                _id : req.params.id
            })

        }
    });
});
```

A continuación, probamos la actualización desde **Postman**, modificando parcialmente una de las canciones.



Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-10.1- SW API REST CRUD Canciones".*

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 los HEADERS (este último es lo más común).

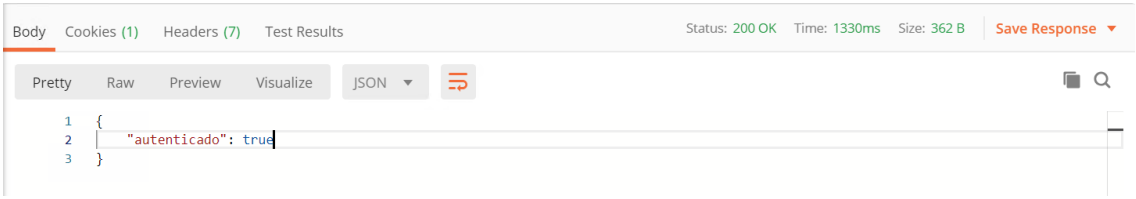
Aunque existen otros mecanismos de identificación basados en tokens la mayoría de APIs REST se basan en los siguientes:

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. De esta forma, la aplicación controla si el usuario tiene permisos o no 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 **/autentificar**, 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, retornamos un JSON con el parámetro autenticado a true o false en caso contrario.

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

    let criterio = {
        email : req.body.email,
        password : seguro
    }

    gestorBD.obtenerUsuarios(criterio, function(usuarios) {
        if (usuarios == null || usuarios.length == 0) {
            res.status(401); // Unauthorized
            res.json({
                autenticado : false
            })
        } else {
            res.status(200);
            res.json({
                autenticado : true
            })
        }

    });
});
```

Devolverá:



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) **encriptamos 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\workspaces\WebstormProjects\sdi1920-lab-node> npm install jsonwebtoken
npm WARN tienda_musica@1.0.0 No description
npm WARN tienda_musica@1.0.0 No repository field.
npm WARN tienda_musica@1.0.0 No license field.

+ jsonwebtoken@8.5.1
added 13 packages from 9 contributors and audited 220 packages in 2.358s
found 3 vulnerabilities (2 low, 1 high)
  run `npm audit fix` to fix them, or `npm audit` for details
PS C:\Dev\workspaces\WebstormProjects\sdi1920-lab-node>
```

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).

```
var express = require('express');
var app = express();

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

Modificamos la respuesta de la petición **POST** a **/api/autenticar** que implementamos en **raplicaciones.js** . Cuando las credenciales sean correctos generamos un token basado en el email del usuario el timestamp en segundos (Date.now() proporciona milisegundos, por lo que debemos dividirlo entre 1000). **Retornamos el nuevo token como respuesta**.

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

    var criterio = {
        email : req.body.email,
        password : seguro
    }

    gestorBD.obtenerUsuarios(criterio, function(usuarios) {
        if (usuarios == null || usuarios.length == 0) {
            res.status(401);
            res.json({
                autenticado : false
            })
        } else {
            var token = app.get('jwt').sign(
                    {usuario: criterio.email , tiempo: Date.now()/1000},
                    "secreto");
            res.status(200);
            res.json({
                autenticado: true,
                token : token
            });
        }

    });
});
```

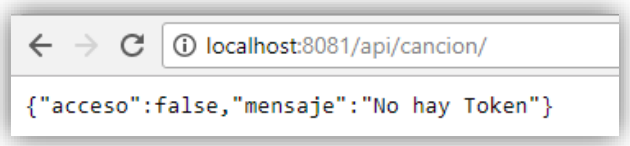
Vamos a requerir el uso de este token para cualquier servicio de **/api/cancion/,** por lo que implementaremos un **nuevo Router (routerUsuarioToken)** en el fichero principal **app.js:**

1. **Obtenemos el parámetro token**: admitimos que se envíe como parámetro: POST, GET o HEADER.
2. **Verificamos el token desencriptándolo**:
    a. **Si no conseguimos desencriptarlo o si han pasado más de 240 segundos** desde que se creó el token retornamos 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.usuario** , de esta forma no habrá que volver a des encriptar el token.

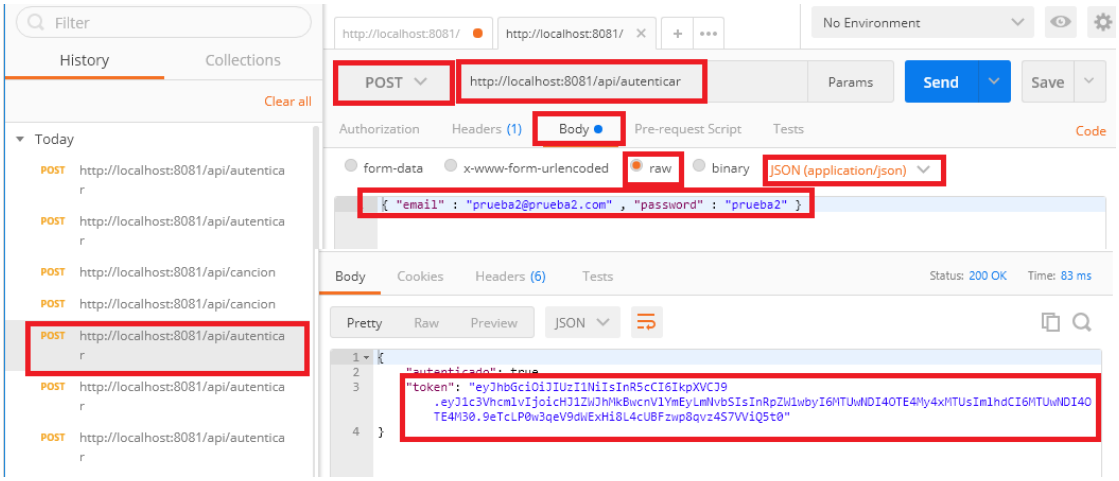
3. **Si no hay token, enviamos un mensaje de error** "No hay token".

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));

// routerUsuarioToken
var routerUsuarioToken = express.Router();
routerUsuarioToken.use(function(req, res, next) {
          // obtener el token, vía headers (opcionalmente GET y/o POST).

          var 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.tiempo) > 240 ){
                           res.status(403); // Forbidden
                           res.json({
                                   acceso : false,
                                   error: 'Token invalido o caducado'
                           });
                           // También podríamos comprobar que intoToken.usuario existe
                           return;

                       } else {
                         // dejamos correr la petición
                         res.usuario = infoToken.usuario;
                         next();
                       }
                   });

          } else {
                     res.status(403); // Forbidden
                     res.json({
                         acceso : false,
                         mensaje: 'No hay Token'
                     });
          }
});

// Aplicar routerUsuarioToken
app.use('/api/cancion', routerUsuarioToken);

// routerUsuarioSession
var routerUsuarioSession = express.Router();
routerUsuarioSession.use(function(req, res, next) {
         console.log("routerUsuarioSession");
          if ( req.session.usuario ) {
            // dejamos correr la petición
             next();
          } else {
             req.session.destino = req.originalUrl;
             console.log("va a : "+req.session.destino)
             res.redirect("/identificarse");
          }
});
```

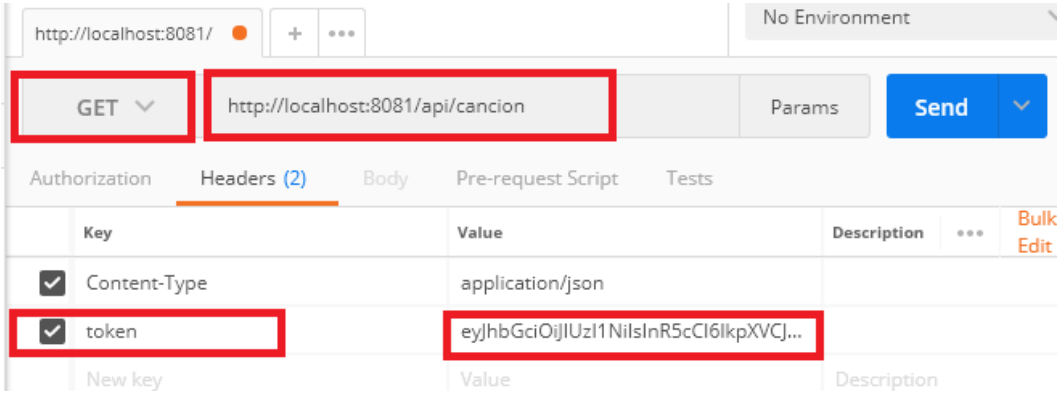
Guardamos los cambios, ejecutamos la aplicación e intentamos a acceder al servicio desde el navegador: https://localhost:8081/api/cancion .



Para probar en detalle todo lo implementado necesitaremos utilizar **PostMan**, primero hacemos una petición **POST** a https://localhost:8081/api/autenticar (Incluyendo en el body el email y password), **si la petición es correcta nos retornará el token**.



Copiamos el token y creamos una nueva petición GET dirigida a https://localhost:8081/api/cancion , **incluimos una nueva cabecera Headers, con clave token y como valor el token obtenido**.



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: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-10.2- SW API REST Autenticación".*

Parte 2 - Cliente jQuery-Ajax

Access-Control-Allow-Origin

Por defecto y por razones de seguridad la aplicación **tienda_musica 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.

```
// Módulos
let express = require('express');
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 de tienda_musica**. 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**, siendo la **combinación jQuery + AJAX la más popular**. Esta combinación será la que utilicemos puesto que, 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, véase Apache). En este caso para simplificar el desarrollo vamos a hacer que esta aplicación sea parte de nuestra aplicación anterior **tienda_musica**.

Descargamos el fichero **cliente.html** del campus virtual. Esta página será la **base de un cliente jQuery que utilizará los servicios web REST de tienda_musica**. **Copiamos cliente.html en el directorio /public/ del proyecto tienda_musica.** Si teníamos más ficheros .html en el directorio **/public** los eliminamos.

Verificamos que contiene los scripts de **jquery** y **bootstrap** necesarios.

```
<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>
```

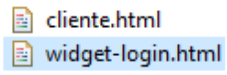
A través de cliente.html cargaremos dinámicamente diferentes componentes/widgets (un widget con el formulario de identificación, otro para mostrar las canciones, etc.) **que se mostrarán dentro del con id="contenedor-principal**".

Agregamos un script al final del fichero en el que incluiremoslas variables generales 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="contenedor-principal"> <!--  id para identificar -->

</div>

<script>

var token;
var URLbase = "https://localhost:8081/api";

$( "#contenedor-principal" ).load( "widget-login.html");

</script>
</body>
```

Sistema de login

Creamos un nuevo fichero **widget-login.html** en la carpeta **/public/.**



Insertamos 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 click en el **botón-login**. Como resultado, se envía el contenido de los campos **email** y **password** al servicio **POST en https://localhost:8081/api/autenticar .**

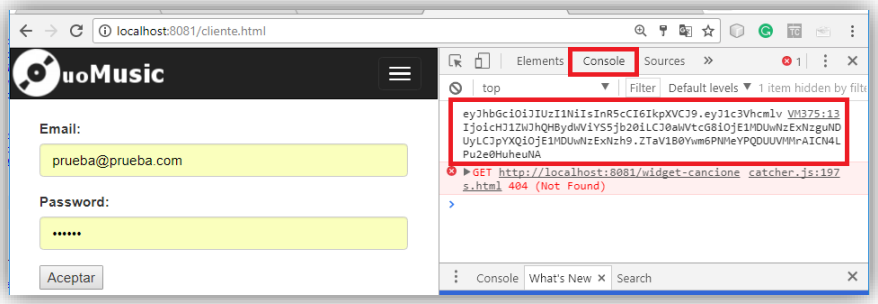
La petición se realiza a través del objeto $.ajax, esperando una respuesta en el formato especificado por **dataType** (JSON en nuestro caso). Concretamente, debería **devolvernos un objeto JSON con la propiedad token, cuyo valor almacenaremos** para futuras peticiones.

Una vez realizada la autenticación con éxito**, vamos a mostrar al usuario automáticamente la lista de canciones**. Como los códigos de respuesta http del servicio están bien implementados (realizados en la parte 1) la petición sabe gestionar los **success (200)** y los **error (401).**

```
<script>

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

</script>
```

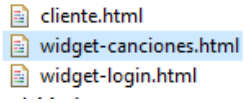
Es importante que el fragmento de <script> se añada después del HTML. En caso contrario, estamos intentando registrar un evento **click()** en un botón **botón-login** que aún no está creado. Otra opción para estar seguros de que los JS se ejecutan cuando toda la web está cargada es usar el *window.onload*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

Accedemos a **https://localhost:8081/cliente.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.



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-canciones.html**



Comenzamos definiendo la **vista HTML**, definiendo una tabla con el cuerpo vacío. Es importante **identificar el cuerpo de la tabla con una id** ya que será donde **agregaremos las canciones**.

```
<div id="widget-canciones" >
    <button class="btn" onclick="cargarCanciones()" >Actualizar</button>
    <table class="table table-hover">
        <thead>
            <tr>
                <th>Nombre</th>
                <th>Genero</th>
                <th>Precio</th>
                <th class="col-md-1"></th>
            </tr>
        </thead>
        <tbody id="tablaCuerpo">

        </tbody>
    </table>
</div>
```

A continuación de la vista incluimos el script, definimos la función **cargarCanciones()**. Esta función hará una petición **GET** a **/cancion** para obtener las canciones e insertarlas en el cuerpo de la tabla (**td=tablaCuerpo**) a través de la función **actualizarTabla(canciones)** .

Invocamos a la función principal **cargarCanciones()** para que se carguen cuando se habrá la página.

```
<script>

var canciones;

function cargarCanciones(){
    $.ajax({
        url: URLbase + "/cancion",
        type: "GET",
        data: { },
        dataType: 'json',
        headers: { "token": token },
        success: function(respuesta) {
            canciones = respuesta;
            actualizarTabla(canciones);
        },
        error : function (error){
            $( "#contenedor-principal" ).load("widget-login.html");
        }
     });
}


function actualizarTabla(cancionesMostrar){
```

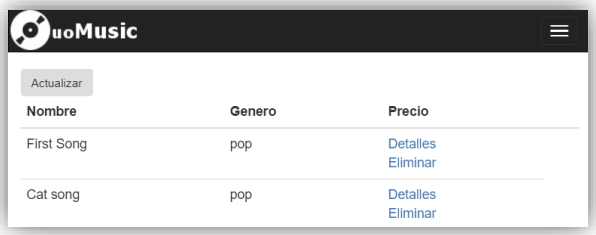
```
    $( "#tablaCuerpo" ).empty(); // Vaciar la tabla
    for (i = 0; i < cancionesMostrar.length; i++) {
        $( "#tablaCuerpo" ).append(
                "<tr id="+cancionesMostrar[i]._id+">"+
                "<td>"+cancionesMostrar[i].nombre+"</td>" +
                "<td>"+cancionesMostrar[i].genero+"</td>" +
                "<td>"+cancionesMostrar[i].precio+"</td>" +
                "<td>"+
                "<a onclick=detalles('"+cancionesMostrar[i]._id+"')>Detalles</a><br>"+
                "<a onclick=eliminar('"+cancionesMostrar[i]._id+"')>Eliminar</a>"+
                "</td>"+
                "</tr>" );
        // Mucho cuidado con las comillas del eliminarCancion
        //la id tiene que ir entre comillas ' '
    }
}

cargarCanciones();

</script>
```

En cada registro de la tabla hemos incluido dos botones con enlaces a las funciones **detalles(id)** y **eliminar(id)**. Debemos de ser muy cuidadosos al hacer **appends** de código HTML que necesite comillas. En el caso del parámetro de las funciones **detalles(id) y eliminar(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 **cargarCanciones**().

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() {
        cargarCanciones();
}, 5000);
```



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

```
function widgetCanciones(){
    $( "#contenedor-principal" ).load( "widget-canciones.html");
}
```

Eliminar canción

Implementaremos la función **eliminar (_id)** en **widget-canciones.html.** Si la petición de eliminar se completa con éxito **(success)**, eliminamos el **<tr>** correspondiente a la canción (también podríamos volver a llamar a **cargarCanciones()** y recargar toda la tabla).

```
function eliminar( _id ) {

    $.ajax({
         url: URLbase + "/cancion/"+_id,
         type: "DELETE",
         data: { },
         dataType: 'json',
         headers: { "token": token },
         success: function(respuesta) {
             console.log("Eliminada: "+_id);
             $( "#"+_id ).remove(); // eliminar el <tr> de la canción
         },
         error : function (error){
                 $( "#contenedor-principal" ).load("widget-login.html");
         }
      });
}
```

Comprobamos que la acción eliminar funciona de forma correcta.

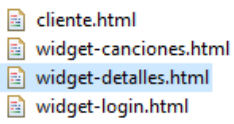
Ver detalles de canción

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

1. Guardar en la variable global **idCancionSeleccionada** la canción seleccionada. Esta variable va utilizarse para intercambiar datos entre widgets, anteriormente habíamos hecho algo similar con la variable **token** (no es estrictamente necesario declarar la variable **idCancionSeleccionada** en ningún sitio, aunque si podríamos declararla en cliente.html).
2. Cargar la vista **widget-detalles.html**, la cual implementaremos posteriormente.

```
cargarCanciones();

function detalles(_id) {
        idCancionSeleccionada = _id;
        $( "#contenedor-principal" ).load( "widget-detalles.html");
}
```

Para mostrar los detalles crearemos el fichero widget-detalles.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="detalles-nombre">Nombre:</label>
      <div class="col-sm-10">
        <input type="text" class="form-control" name="detalles-nombre"
            placeholder="Nombre de mi canción"  id="detalles-nombre" readonly/>
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-sm-2" for="detalles-genero">Genero:</label>
      <div class="col-sm-10">
        <input type="text" class="form-control" name="detalles-genero"
            placeholder="Nombre de mi canción"  id="detalles-genero" readonly/>
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-sm-2" for="detalles-precio">Precio (€):</label>
      <div class="col-sm-10">
        <input type="number" step="0.01" class="form-control" name="detalles-precio"
            placeholder="2.50" id="detalles-precio"  readonly/>
      </div>
    </div>
    <button onclick="widgetCanciones()" class="btn" >Volver</button>
</div>
```

A continuación, vamos a incluir en **widget-detalles.html** el script: supondremos que la variable **_idCancionSeleccionada** 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 + "/cancion/" + idCancionSeleccionada ,
    type : "GET",
    data : {},
    dataType : 'json',
    headers : {
        "token" : token
    },
    success : function(cancion) {
        $("#detalles-nombre").val(cancion.nombre);
        $("#detalles-genero").val(cancion.genero);
        $("#detalles-precio").val(cancion.precio);
    },
    error : function(error) {
        $( "#contenedor-principal" ).load("widget-login.html");
    }
});

</script>
```

Comprobamos que la aplicación muestra los detalles de la canción correctamente.

Agregar canción

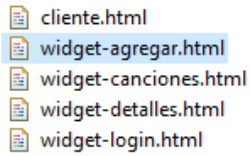
Abrimos el fichero **widget-canciones.html** y agregamos un nuevo botón en su vista HTML.

```
<div id="widget-canciones" >
    <button class="btn btn-primary" onclick="widgetAgregar()">Nueva Canción</button>
    <button class="btn" onclick="cargarCanciones()" >Actualizar</button>
        <table class="table table-hover">
```

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

```
function widgetAgregar() {
        $( "#contenedor-principal" ).load( "widget-agregar.html");
}
```

Creamos el fichero **widget-agregar.html** en la carpeta **/public.**



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

```
<div id="widget-agregar" >
        <div class="form-group">
      <label class="control-label col-sm-2" for="agregar-nombre">Nombre:</label>
      <div class="col-sm-10">
        <input type="text" class="form-control" name="agregar-nombre"
            placeholder="Nombre de mi canción"  id="agregar-nombre" />
      </div>
    </div>
        <div class="form-group">
      <label class="control-label col-sm-2" for="agregar-genero">Genero:</label>
      <div class="col-sm-10">
        <input type="text" class="form-control" name="agregar-genero"
            placeholder="Nombre de mi canción"  id="agregar-genero" />
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-sm-2" for="agregar-precio">Precio (€):</label>
      <div class="col-sm-10">
        <input type="number" step="0.01" class="form-control" name="detalles-precio"
        placeholder="2.50" id="agregar-precio"  />
      </div>
    </div>
    <div class="col-sm-offset-2 col-sm-10">
        <button type="button" class="btn btn-primary" id="boton-agregar"
           onclick="agregarCancion()">Agregar</button>
    </div>
</div>
```

Implementamos la función **agregarCanción()** dentro de **widget-agregar.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 **https://localhost:8081/api/cancion** .
2. Una vez agregada la canción, vuelve a mostrar la lista de las canciones.

```
<script>
function agregarCancion( ) {
    $.ajax({
          url: URLbase + "/cancion",
          type: "POST",
          data: {
              nombre : $("#agregar-nombre").val(),
              genero : $("#agregar-genero").val(),
              precio  : $("#agregar-precio").val()
          },
          dataType: 'json',
          headers: { "token": token },
          success: function(respuesta) {
             console.log(respuesta); // <-- Prueba
             $( "#contenedor-principal" ).load( "widget-canciones.html");

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

Ejecutamos la aplicación y comprobamos que nos permite agregar canciones.

Nota: Subir el codigo a GitHub en este punto. Commit Message → *"SDI-NodeJS-10.3- SW Cliente jQuery + AJAX".*

Ampliación - Filtrado dinámico

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

```
<div id="widget-canciones" >
    <input type="text"  class="form-control"  placeholder="Filtrar por nombre"
id="filtro-nombre"/>
    <button class="btn btn-primary" onclick="widgetAgregar()">Nueva Cancion</button>
    <button class="btn" onclick="cargarCanciones()" >Actualizar</button>
```

En la lógica **implementaremos un listener sobre el input con id filtro-nombre** para que, cada vez que se detecte un cambio, recorra la lista de canciones originales y guarde en el array **cancionesFiltradas** las canciones que tienen coincidencia con el texto introducido. Finalmente representamos las canciones presentes en el array **cancionesFiltradas**.

```
$('#filtro-nombre').on('input',function(e){
    var cancionesFiltradas = [];
    var nombreFiltro = $("#filtro-nombre").val();

    for (i = 0; i < canciones.length; i++) {
        if (canciones[i].nombre.indexOf(nombreFiltro) != -1 ){
            cancionesFiltradas.push(canciones[i]);
        }
    }
    actualizarTabla(cancionesFiltradas);
});
```

Ampliación - Ordenación

En el propio **widget-canciones.html** incluiremos un sistema de ordenación por precio y nombre. Incluimos en la cabecera de la tabla para Nombre y Precio los enlaces que invocarán respectivamente a **ordenarPorNombre()** y **ordenarPorPrecio().**

```
<thead>
    <tr>
        <th><a onclick="ordenarPorNombre()">Nombre</a></th>
        <th>Genero</th>
        <th><a onclick="ordenarPorPrecio()">Precio</a></th>
        <th class="col-md-1"></th>
    </tr>
</thead>
```

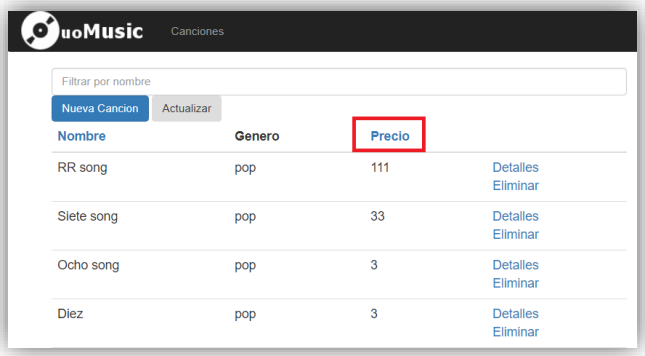
Implementamos la función **ordenarPorPrecio() que ordenará el array utilizamos la función sort**. Dependiendo del retorno de la función de ordenación colocará una canción (a y b) delante de otra:

- >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 ordenarPorPrecio(){
    canciones.sort(function(a, b) {
        if (parseFloat(a.precio) > parseFloat(b.precio)) return -1;
        if (parseFloat(a.precio) < parseFloat(b.precio)) return 1;
        return 0
        // Dependiendo de retorno >0 =0 o <0
    });
    actualizarTabla(canciones);
}

function ordenarPorNombre(){
    canciones.sort(function(a, b) {
         if(a.nombre > b.nombre ) return 1;
         if(a.nombre < b.nombre ) return -1;
         return 0;
    });
    actualizarTabla(canciones);
}
```

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



Si quisiésemos que **con cada click sobre el enlace se invirtiera la ordenación**, bastaría con guardar un boolean con el estado de cada orden y, en función del estado, elegir un orden u otro para invertir el estado.

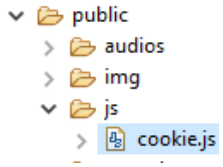
```
var precioDsc = true;

function ordenarPorPrecio(){
    if (precioDsc){
        canciones.sort(function(a, b) {
            return parseFloat(a.precio) - parseFloat(b.precio);
        });
```

```
    } else {
        canciones.sort(function(a, b) {
            return parseFloat(b.precio) - parseFloat(a.precio);
        });
    }
    actualizarTabla(canciones);
    precioDsc = !precioDsc; //invertir
}
```

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.

JS 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 campus virtual y lo copiamos en la carpeta **/public/js/ .**



Agregamos la referencia al fichero **cookie.js** en el fichero principal **cliente.html.**

```
<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>
<script src="/js/cookie.js"></script>
```

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

En el script del fichero **cliente.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-canciones**.

```
var token;
var URLbase = "https://localhost:8081/api";

$( "#contenedor-principal" ).load( "widget-login.html");

if ( Cookies.get('token') != null ){
        token = Cookies.get('token');
        $( "#contenedor-principal" ).load( "widget-canciones.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 **https://localhost:8081/cliente.html** , esto **imposibilita la compartición de URLs a partes concretas de la aplicación**. Aunque esto puede no ser un problema en algunas aplicaciones nos podrían requerir esta funcionalidad.

Debemos implementar un **sistema de URLs empleando parámetros GET**. En este caso vamos a manejar dos posibilidades **cliente.html?w=login** , **cliente.html?w=canciones** y las que fueran necesarias.

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

```
<script>
window.history.pushState("", "", "/cliente.html?w=login");
```

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

```
<script>
window.history.pushState("", "", "/cliente.html?w=canciones");
```

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

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

```
<script>
var token;
var URLbase = "https://localhost:8081/api";

$( "#contenedor-principal" ).load( "widget-login.html");

if ( Cookies.get('token') != null ){
    token = Cookies.get('token');

    var url = new URL(window.location.href);
    var w = url.searchParams.get("w");
    if ( w == "login"){
        $( "#contenedor-principal" ).load("widget-login.html");
    }
    if ( w == "canciones"){
        $( "#contenedor-principal" ).load("widget-canciones.html");
    }
}
```

Probamos el funcionamiento de las URLs: https://localhost:8081/cliente.html?w=login y **https://localhost:8081/cliente.html?w=canciones**

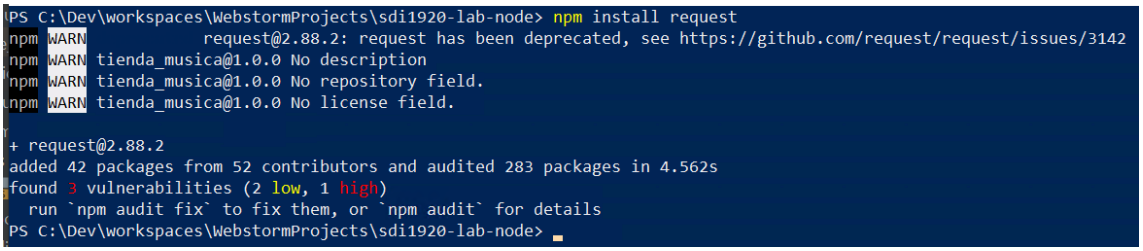
Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-10.4- SW Cliente jQuery + AJAX (Ampliaciones)".*

Parte 3 – Consumir SW REST desde Node JS

Cliente REST en Node JS

Vamos a **implementar un cliente REST dentro de la aplicación tienda_musica**. Este cliente va a utilizar el servicio web REST comercial https://api.exchangeratesapi.io/ que permite hacer transformaciones de divisas en base a la cotización real.

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**.



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'.

```
// Módulos
let express = require('express');
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://api.exchangeratesapi.io/latest?base=EUR** que, según la documentación nos retornará un objeto JSON.

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

En el **controlador rcanciones.js buscaremos y modificaremos la función que responde a GET /cancion/: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 exchangerateapi**.

Debemos declarar un objeto de configuración especificando la información de la petición a realizar: **url, method, headers** (en caso de que tenga una header/cabecera, hemos incluido una a modo de ejemplo, pero realmente el servicio de 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 retorna https://exchangeratesapi.io/ aunque sepamos que la respuesta está en formato JSON debemos convertirla ya que se procesa inicialmente como un String.

El objeto respuesta contiene atributos con claves: base, date, rates, siendo la variable rates de tipo objeto { } y teniendo una variable con nombre USD de tipo numérico.

```
{"base":"EUR","date":"2019-03-27","rates":{"USD":1.1235}}
```

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 bcancion**.

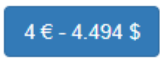
Nota: *Si hemos modificado la aplicación con algunos de los ejercicios complementarios* es posible que esta función sea necesario adaptarla al código de la versión que tengamos.

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

    gestorBD.obtenerCanciones(criterio, function (canciones) {
        if (canciones == null) {
            res.send(respuesta);
        } else {
            var configuracion = {
                url: "https://api.exchangeratesapi.io/latest?base=EUR",
                method: "get",
                headers: {
                    "token": "ejemplo",
                }
            }
            var rest = app.get("rest");
            rest(configuracion, function (error, response, body) {
                console.log("cod: " + response.statusCode + " Cuerpo :" + body);
                var objetoRespuesta = JSON.parse(body);
                var cambioUSD = objetoRespuesta.rates.USD;
                // nuevo campo "usd"
                canciones[0].usd = cambioUSD * canciones[0].precio;
                var respuesta = swig.renderFile('views/bcancion.html',
                    {
                        cancion: canciones[0]
                    });
                res.send(respuesta);
            })
            var respuesta = swig.renderFile('views/bcancion.html',
            {
                cancion : canciones[0]
            });

        }
    });
```

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

```
/{{ cancion._id.toString() }}">{{ cancion.precio }} € - {{ cancion.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 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
    var anuncio = {
        descripcion : 'Nuevo anuncio',
        precio : '10'
    };
    var configuracion = {
        url: "http://ejemplo.ejemplo/anuncio",
        method: "POST",
        json: true,
        headers: {
            "content-type": "application/json",
        },
        body: anuncio
    }
    var rest = app.get("rest");
    rest(configuracion, function (error, response, body) {
    ...
    })
```

Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-10.5- SW Consumir SW desde NodeJS".*

Parte 4 – Consumir SW REST desde Cliente Java

En este apartado, utilizaremos tanto el Webstorm (para crear un servicio web en NodeJS) como el STS (u otro IDE Java) para crear un cliente en JAVA para consumir el servicio.

Creando el Servicio Web

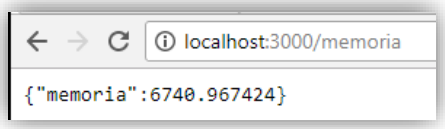
En la práctica 7 (primera práctica de Node), creamos un **fichero hello-world-server.js** cuyo contenido modificaremos para **crear un servicio web que responda a la petición GET /memoria**. Este servicio nos devolverá un JSON con la memoria libre en el equipo.

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, por ejemplo, que maneje muchos datos.

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

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

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

    }, 10000);
});

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

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



Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-10.6- SW Servicio Memoria".*

Creando el cliente Java

Como primer paso, **crearemos un nuevo repositorio en GitHub con el nombre sdi1920-x-lab-swjava (sustituyendo la x por el IDGIT correspondiente)**.

¡¡¡¡¡¡MUY IMPORTANTE!!!!!!

Aunque en este guión se ha usado como nombre de repositorio para todo el ejercicio **sdi1920-lab-swjava**, cada alumno deberá usar como nombre de repositorio **sdi1920-x-lab-swjava**, donde **x** = columna **IDGIT** en el Documento de ListadoAlumnosSDI1920.pdf del CV**.**

Por ejemplo el alumno con IDGIT=101, deberá crear un repositorio con nombre sdi1920-101-lab-node y un proyecto node en el IDE con nombre también sdi1920-101-lab-node.

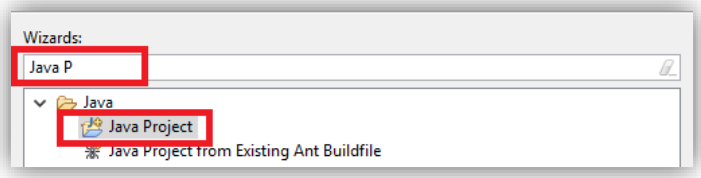
En resumen:

Nombre repositorio GitHub: **sdi1920-x-lab-swjava**

Nombre proyecto STS: **sdi1920-x-lab-swjava**

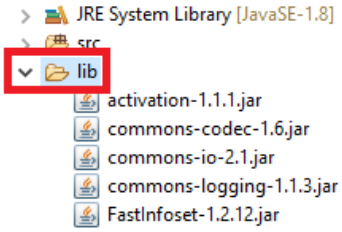
Otra cuestión **IMPRESCINDIBLE** es que una vez creado el repositorio deberá invitarse como colaborador a dicho repositorio a la cuenta github denominada **sdigithubuniovi.**

En el STS importaremos el proyecto a través de GIT (**Import → Git → Projects from Git → Clone URI**) tal y como hicimos en prácticas anteriores. Una vez más, **emplearemos la opción "Import using the New Project wizard"** **pero esta vez como Java Project**. Como ya se comentó, el proyecto se denominará sdi1920-x-lab-swjava.



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).

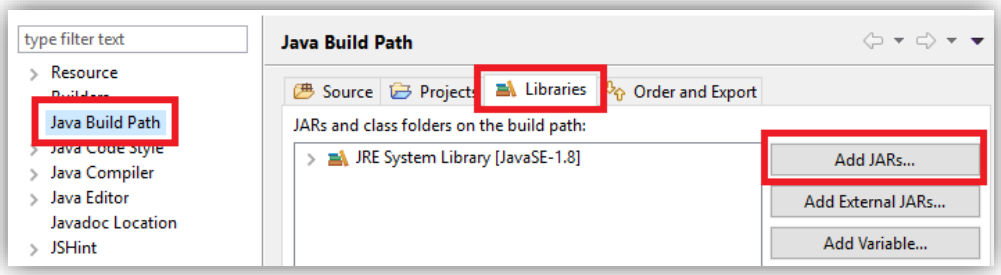
Creamos un nuevo Folder en el proyecto y lo llamamos **lib**, copiamos en el directorio las librerías que nos descargamos del campus virtual. Entre ellas se encuentra **JAX-RS** y otras utilidades.



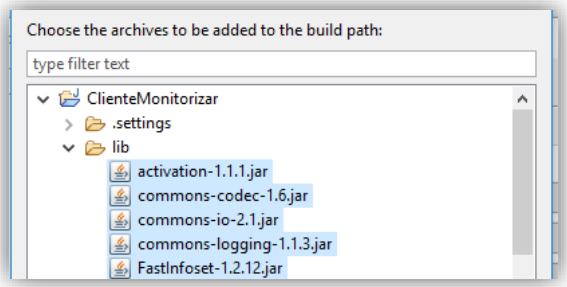
Agregamos al **build path** del proyecto todas las librerías de la carpeta **/lib**

Botón derecho sobre el **proyecto -> properties.**

Seleccionamos en el menú de la izquierda la sección **Java Build Path,** después la pestaña **Libraries** y pulsamos el botón **Add JARs…**



Seleccionamos todos los **.jar** de la carpeta **/lib**



Finalmente pulsamos en el botón **Apply and Close**.

Creamos una nueva clase Java llamada **Ventana** en el paquete **com.sdi**

Implementamos una ventana con **swing** en la que definimos:

- 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.

Incluimos la función **main** en la cual creamos una instancia de la **Ventana**.

```
package com.sdi;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import javax.swing.border.EmptyBorder;
import javax.ws.rs.client.ClientBuilder;
import org.codehaus.jackson.node.ObjectNode;

public class Ventana {
        JFrame frame;
        JPanel panel;
        JButton botonActualizar;
        JButton botonApagar;
        public JLabel textoMemoria;
        int peticiones = 0;
```

```
        public static void main(String[] args) throws InterruptedException {
                new Ventana();
        }
        public Ventana() {
                // 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
                botonActualizar = new JButton("Actualizar Memoria");
                botonActualizar.setBorder(new EmptyBorder(10, 10, 10, 10));
                botonActualizar.addActionListener(new ActionListener() {
                        @Override
                        public void actionPerformed(ActionEvent arg0) {

                        }
                });
                panel.add(botonActualizar);

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

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

Si ejecutamos la aplicación deberíamos ver la ventana.

Nos falta por implementar la lógica que se ejecuta al pulsar el **botonActualizar**, deberíamos:

- Hacer una invocación al servicio REST **GET / http://localhost:3000/memoria . (Comprobad que tenéis el servicio web desarrollado en el apartado anterior arrancado).**
- Procesar el objeto JSON recibido {"memoria":6740.967424}
- Mostrar la memoria consumida en el componente **textoMemoria**

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(…))

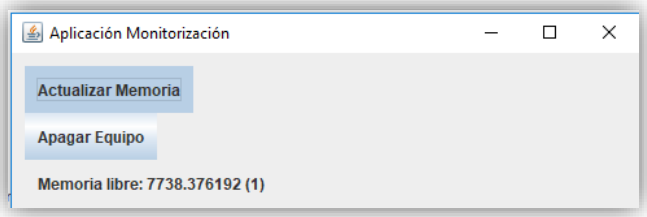
Completamos los parámetros para la petición a nuestro servicio. En el caso del parámetro especificado en la función **readEntity**() utilizaremos un tipo de objeto **ObjectNode**.

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 **RespuestaMemoria** con un atributo **memoria (get/set)** la cual hubiese servido para que se parseara automáticamente el JSON a un objeto de ese tipo)

```
@Override
public void actionPerformed(ActionEvent arg0) {
    peticiones++;

    ObjectNode respuestaJSON;
    respuestaJSON = ClientBuilder.newClient()
            .target("http://localhost:3000/memoria")
            .request()
            .accept(MediaType.APPLICATION_JSON)
            .get()
            .readEntity(ObjectNode.class);

    String memoria = respuestaJSON.get("memoria").toString();
    textoMemoria.setText("Memoria libre: "+memoria +" ("+peticiones+")");
}
```

Ejecutamos la aplicación y comprobamos lo que sucede al pulsar el botón **Actualizar Memoria**.

Durante los 10 segundo que tarda en completarse la petición la aplicación está completamente 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 **ActualizarMemoriaThread**, en esta clase implementamos un hilo (clase que extiende hereda de Thread).

Creamos un constructor donde reciba una referencia a **Ventana** 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 **actualizarMemoria(memoria);** del objeto **Ventana** la cual está aún sin implementar pero debería actualizar la memoria que se muestra por pantalla.

```
public class ActualizarMemoriaThread extends Thread {
    Ventana ventana;

    public ActualizarMemoriaThread(Ventana ventana) {
        this.ventana = ventana;
    }

    public void run(){
        ObjectNode respuestaJSON;
        respuestaJSON = ClientBuilder.newClient()
                .target("http://localhost:3000/memoria")
                .request()
                .accept(MediaType.APPLICATION_JSON)
                .get()
                .readEntity(ObjectNode.class);

        String memoria = respuestaJSON.get("memoria").toString();
        ventana.actualizarMemoria(memoria);
     }

}
```

Volvemos a la clase **Ventana**. Sustituimos el código que se ejecuta al pulsar el **botonActualizar**. Crearemos una instancia del hilo **ActualizarMemoriaThread** y lo lanzaremos (función **start()**)

```
@Override
public void actionPerformed(ActionEvent arg0) {
    peticiones++;
    ActualizarMemoriaThread hilo = new ActualizarMemoriaThread(Ventana.this);
    hilo.start();
}
```

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

```
public  void actualizarMemoria(String memoria) {
   textoMemoria.setText("Memoria libre: "+memoria +" ("+peticiones+")");
}
```

Aunque este código puede que en ocasiones funcione correctamente, si lo utilizamos estaríamos cayendo en un error. **ActualizarMemoria()** 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).**

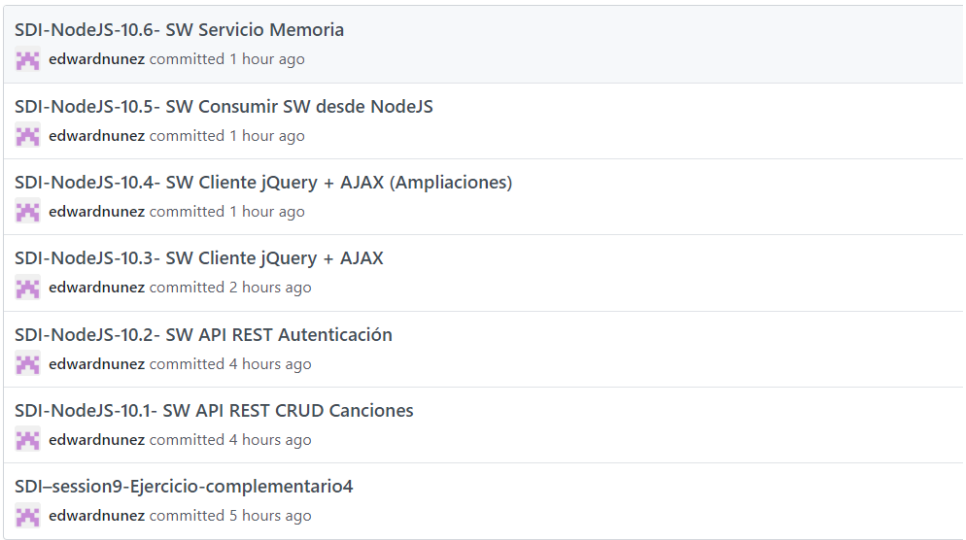
```
public  void actualizarMemoria(String memoria) {
    SwingUtilities.invokeLater(new Runnable(){
        public void run(){
            textoMemoria.setText("Memoria libre: "+memoria +" ("+peticiones+")");
        }
    });
}
```

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

Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-10.7- SW Cliente Java".*

Resultado esperado en los repositorio de GitHub

Proyecto NodeJS



Proyecto JAVA