



Sistemas Distribuidos e Internet

Desarrollo de aplicaciones Web con NodeJS

Sesión - 7

Curso 2019/2020



Contenido

1	Introducción	3
1.1	Configuración del entorno de desarrollo	3
1.1.1	Instalación de Node JS	3
1.1.2	Alternativa RECOMENDADA - IDE WebStorm.....	4
1.1.3	(NO RECOMENDADO) IDE STS: Nodeclipse desde fichero.....	4
1.1.4	(NO RECOMENDADO) IDE STS: Nodeclipse desde Marketplace...	6
1.1.5	(NO RECOMENDADO) IDE STS: Comprobando la instalación	7
1.1.6	(NO RECOMENDADO) IDE STS: Configurando el Workspace.....	8
1.1.7	Creación del repositorio en GitHub	10
2	Creación del proyecto.....	11
3	Express	16
4	Depurar.....	17
5	Variables de entorno.....	19
6	División de las rutas en módulos.....	19
7	Peticiones GET y parámetros	21
8	Recursos estáticos.....	24
9	Peticiones POST y parámetros.....	25
10	Enrutamiento y comodines	27
11	Vistas y Motores de plantillas	27
12	Plantillas – URLs absolutas	32
13	Plantillas – Bloques	33
14	Vista para agregar canciones.....	37
15	Mongo DB en la Nube (Obligatorio)	39
16	Instalación y verificación MongoDB Local (NO HACER).....	43
17	Resultado esperado en el repositorio de GitHub.....	45



1 Introducción

En esta práctica veremos una introducción de cómo desarrollar aplicaciones Web de forma ágil utilizando NodeJS como una plataforma de software para ejecutar JavaScript del lado del servidor.

1.1 Configuración del entorno de desarrollo

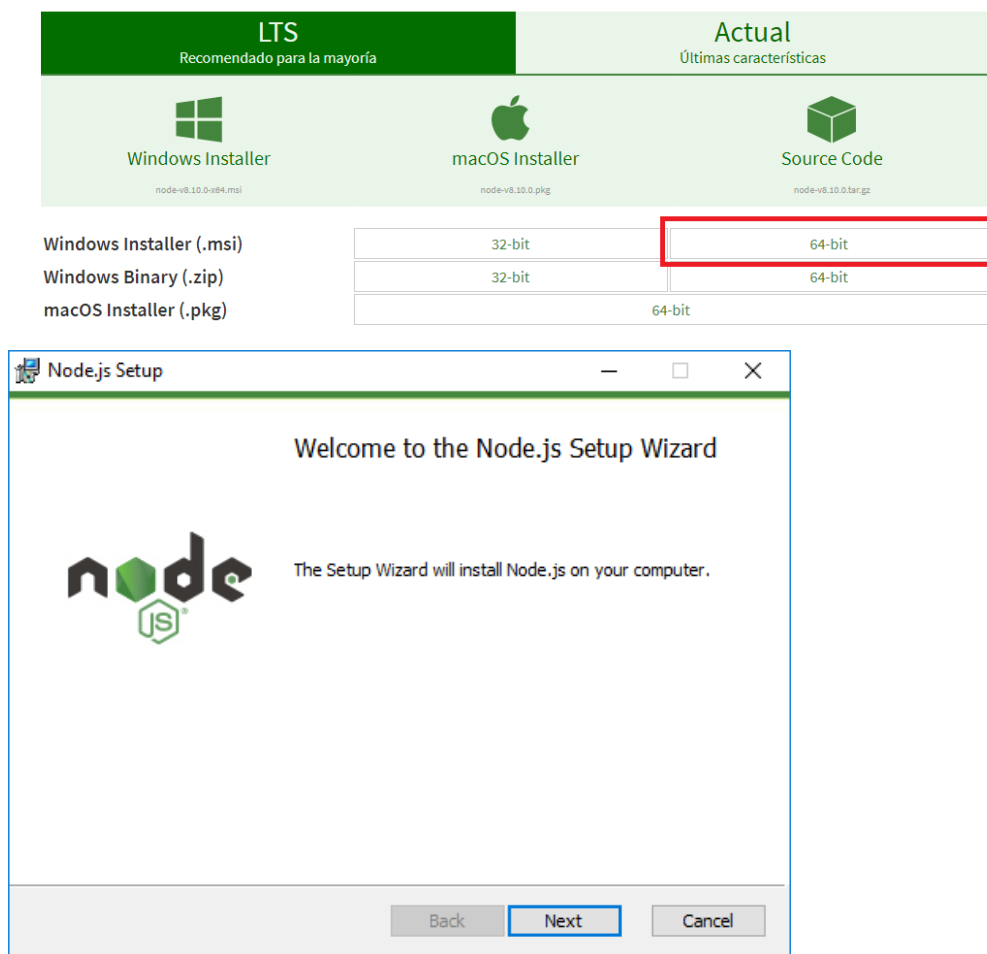
Para el desarrollo de las próximas prácticas utilizaremos un IDE más orientado a JavaScript.

- Como IDE recomendado para ésta y siguientes sesiones prácticas, **recomendamos WebStorm de JetBrains**.
- Otra **opción (no recomendada)** sería utilizar algún IDE basado en Eclipse como **Spring Tool Suite (STS)** añadiéndole un plugin específico (os mostramos cómo instalar un plugin desde fichero y desde el marketplace).

Aunque en el guion os mostremos las dos opciones, NO ES OBLIGATORIO REALIZAR AMBAS. DE HECHO, NOSOTROS OS RECOMENDAMOS WEBSTORM.

1.1.1 Instalación de Node JS

Para trabajar con NodeJs debemos instalarlo. Descargamos el instalador de **Node 12 (Versión 12.*)** o posterior en la siguiente URL <https://nodejs.org/es/download/>



Hacemos click en el botón siguiente y dejamos todo marcado por defecto.

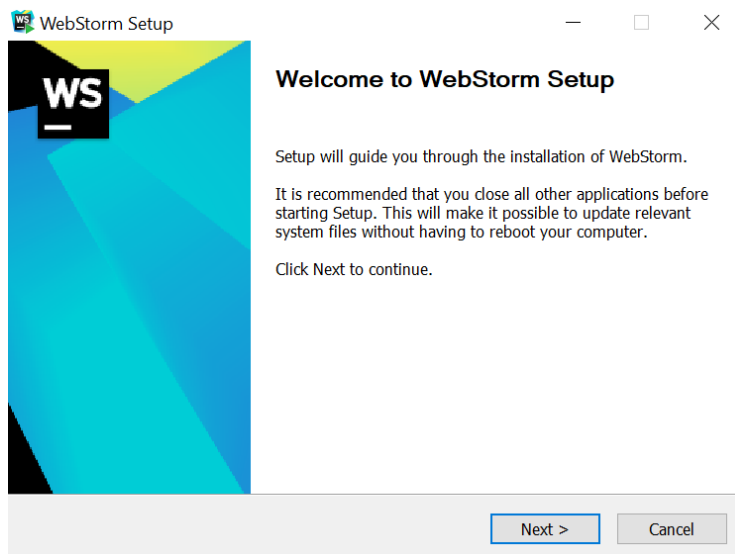


1.1.2 Alternativa RECOMENDADA - IDE WebStorm

Importante: Por defecto, **WebStorm** incluye una prueba gratuita de 30 días. Debéis solicitar una licencia de estudiante (es gratis) en <https://www.jetbrains.com/student/>. Una vez registrados la licencia de estudiante se asocia a vuestro email de la universidad.

Descargaremos el instalador de WebStorm de la siguiente dirección:

<https://www.jetbrains.com/es-es/webstorm/download/>



Seguir los pasos del asistente e instalar el IDE con todas las **características por defecto** o aquellas que se consideren importante para el desarrollo de aplicaciones usando JavaScript.

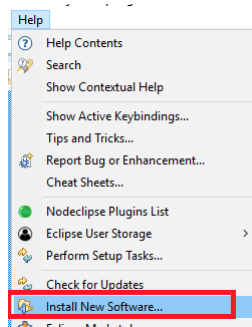
Recuerda: Solicita una licencia de estudiante en <https://www.jetbrains.com/student/>

Si has elegido WebStorm, debes saltar al apartado 1.1.7 del guion.

1.1.3 (NO RECOMENDADO) IDE STS: Nodeclipse desde fichero

¡El IDE recomendado es WebStorm! Ignora este apartado si instalaste WebStorm.

Descargamos el **fichero org.nodeclipse.site...** del campus virtual, **Help → Install new Software**

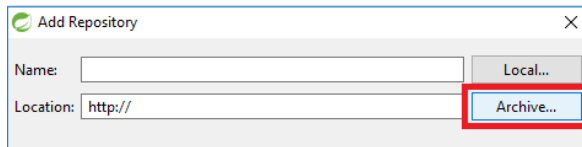




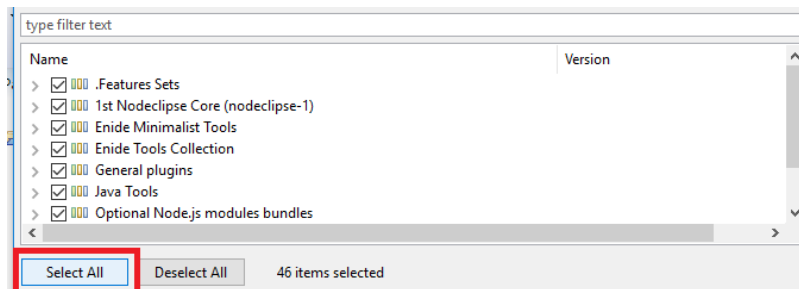
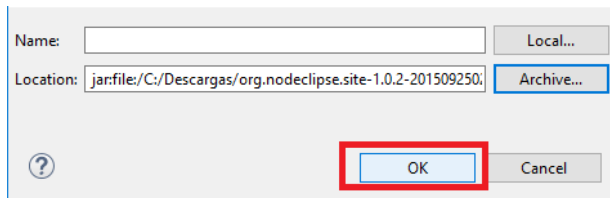
Seleccionamos la opción **Add**



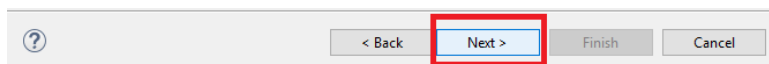
Después indicamos que vamos a utilizar un **Archive** y seleccionamos la ruta del zip descargado previamente.



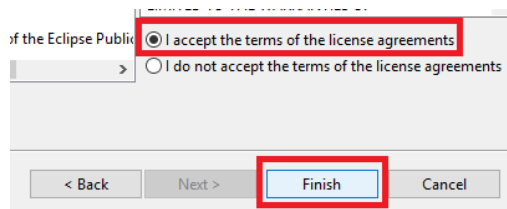
Pulsamos **Ok** y marcamos todo.



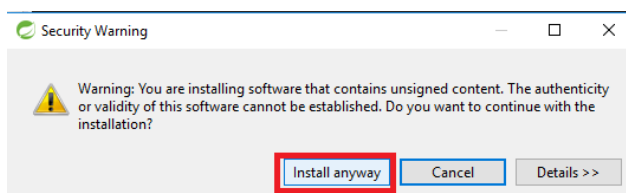
Varias veces durante la instalación nos pedirá darle a **Next >**.

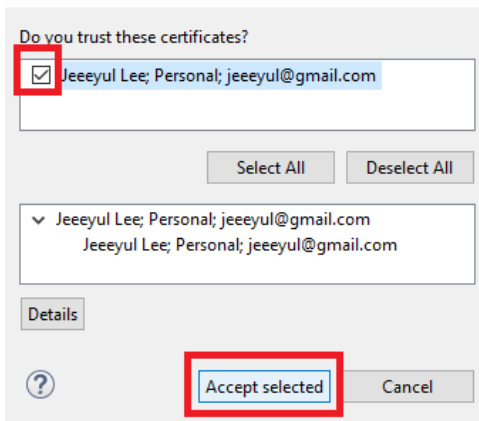


En uno de los últimos pasos debemos aceptar la licencia.



En algún momento nos dirá que la firma no tiene validez pulsamos en **install anyway**.





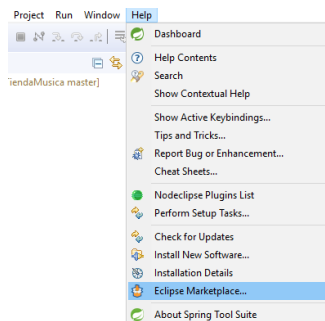
Finalmente nos pedirá reiniciar el IDE y entonces el plugin estará listo para su uso.

A continuación, pasa al apartado 1.1.5 del guion para comprobar la instalación.

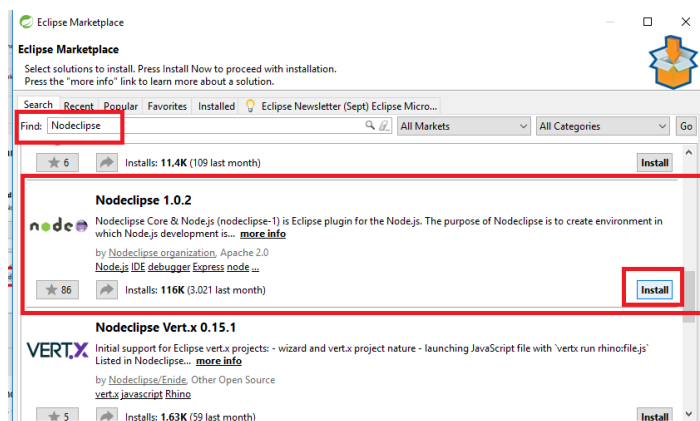
1.1.4 (NO RECOMENDADO) IDE STS: Nodeclipse desde Marketplace

¡El IDE recomendado es WebStorm! Ignora este apartado si instalaste WebStorm.

Si www.nodeclipse.org está activo, el plugin **Nodeclips** se puede descargar del propio Marketplace. Para ello, abrimos la tienda de aplicaciones del eclipse desde **Help → Eclipse Marketplace**.



Utilizamos la búsqueda para encontrar el plugin Nodeclipse.

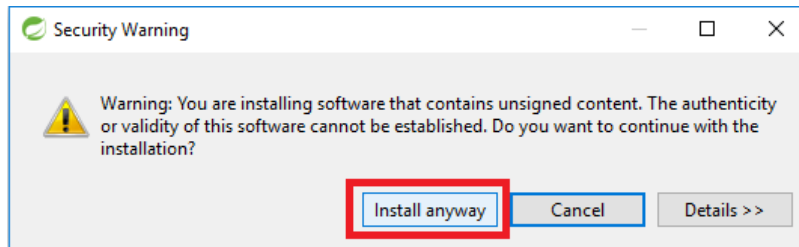




Después de aceptar los términos, comenzará la instalación. Ésta puede tardar varios minutos, pudiendo observar el progreso en la parte inferior derecha de la pantalla.



En algún punto del proceso de instalación es posible que nos pida confirmaciones de seguridad.

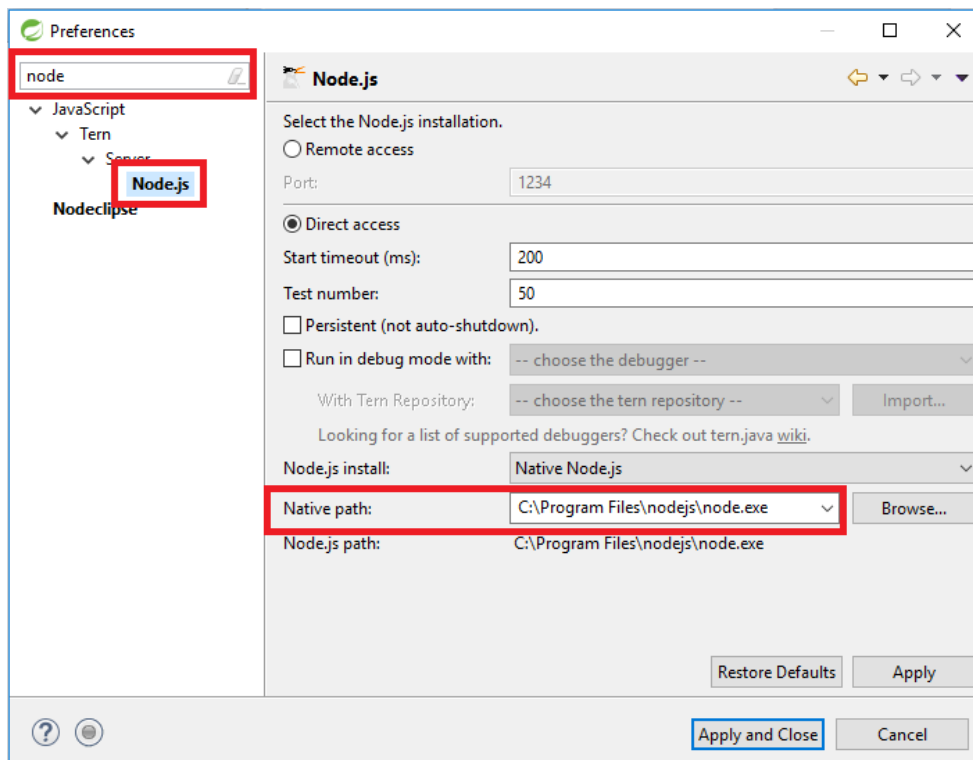


1.1.5 (NO RECOMENDADO) IDE STS: Comprobando la instalación

¡El IDE recomendado es WebStorm! Ignora este apartado si instalaste WebStorm.

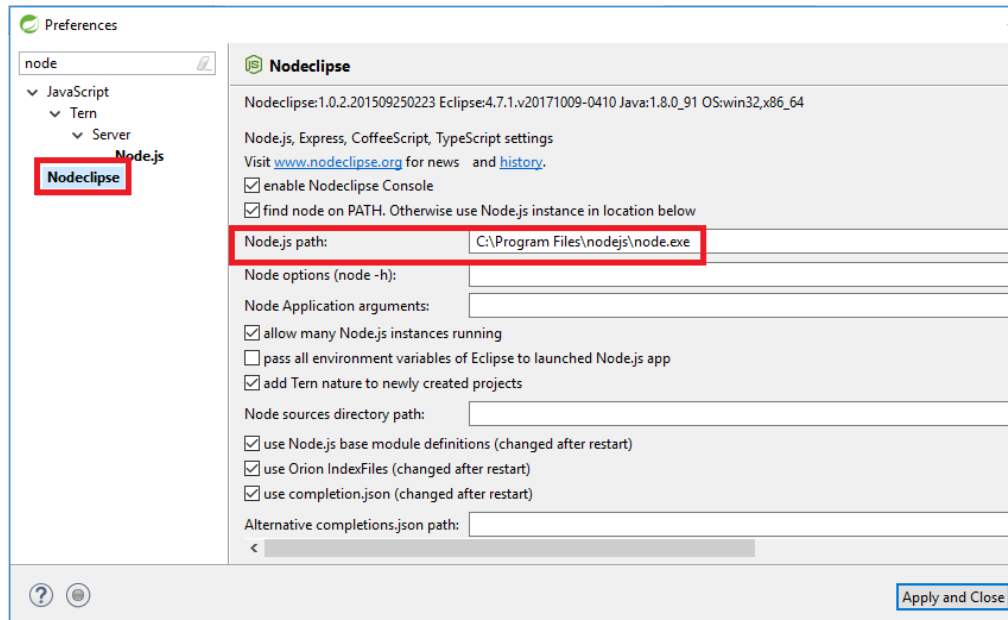
Antes de continuar vamos a verificar que la configuración del entorno es correcta. Accedemos a las preferencias del STS desde la opción de menú **Windows** → **preferences**.

Buscamos la sección **Node.js**, podemos usar el cuadro de búsqueda de la parte superior izquierda para localizarla más rápidamente. Verificamos que el **Native path**, se corresponde con nuestra instalación de Node.





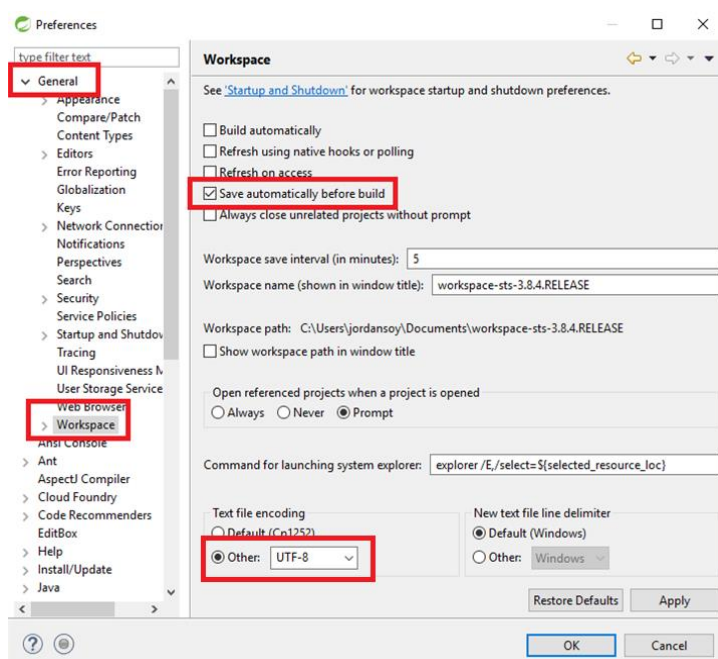
Buscamos también la sección **Nodeclipse** y verificamos que se corresponde con nuestra instalación de node.



1.1.6 (NO RECOMENDADO) IDE STS: Configurando el Workspace

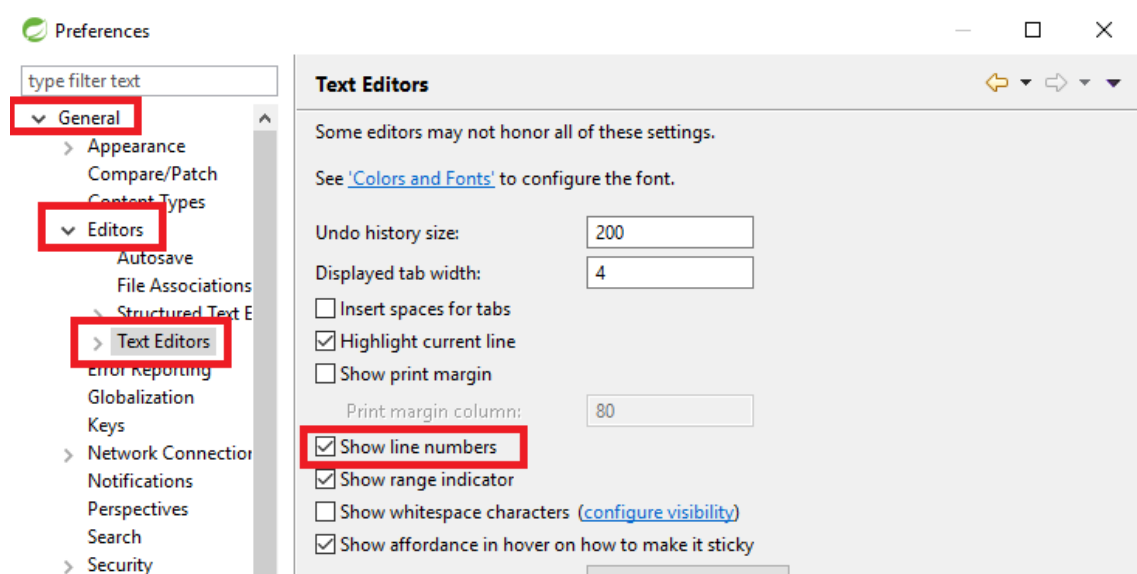
¡El IDE recomendado es WebStorm! Ignora este apartado si instalaste WebStorm.

Antes de comenzar vamos a modificar la configuración de nuestro Workspace. Debemos modificar sus propiedades desde **Window → Preferences → General → Workspace**. Básicamente estableceremos la codificación a **UTF-8**.

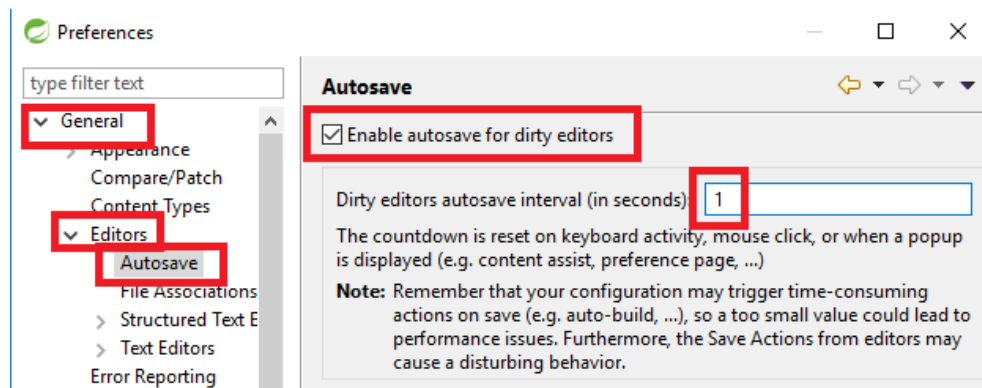




También vamos a **Editor** → **Text Editors**, y marcamos la opción **“Show line numbers”**.



Los ficheros no se salvan automáticamente al desplegar la aplicación. Es recomendable habilitar la opción de Autoguardado: **General** → **Editors** → **Autosave**.





1.1.7 Creación del repositorio en GitHub

En este guion práctico se asume que en anteriores prácticas habéis instalado Git en vuestro equipo y que habéis creado una cuenta en GitHub.

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

Aunque en este guion se ha usado como nombre de repositorio para todo el ejercicio **sdi-lab-node**, cada alumno deberá usar como nombre de repositorio **sdix-lab-node**, 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 sdi101-lab-node y un proyecto node en el IDE con nombre también sdi101-lab-node.

En resumen:

Nombre repositorio GitHub: **sdix-lab-node**

Nombre proyecto WebStorm: **sdix-lab-node**

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

Creamos un **nuevo Repositorio** en GitHub al que denominaremos **sdix-lab-node**. Este repositorio es el que utilizaremos para completar la práctica. Para ello, desde la página Web de **GitHub** buscamos la sección de repositorios y hacemos click en el botón **New** (<https://github.com/new>).

Create a new repository

A repository contains all project files, including the revision history.

Owner: edwardnunez / Repository name: sdi-lab-node

Great repository names are s. Your new repository will be created as sdi-lab-node- friendly-funicular?

Description (optional):

☒ Public
Anyone can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

☒ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

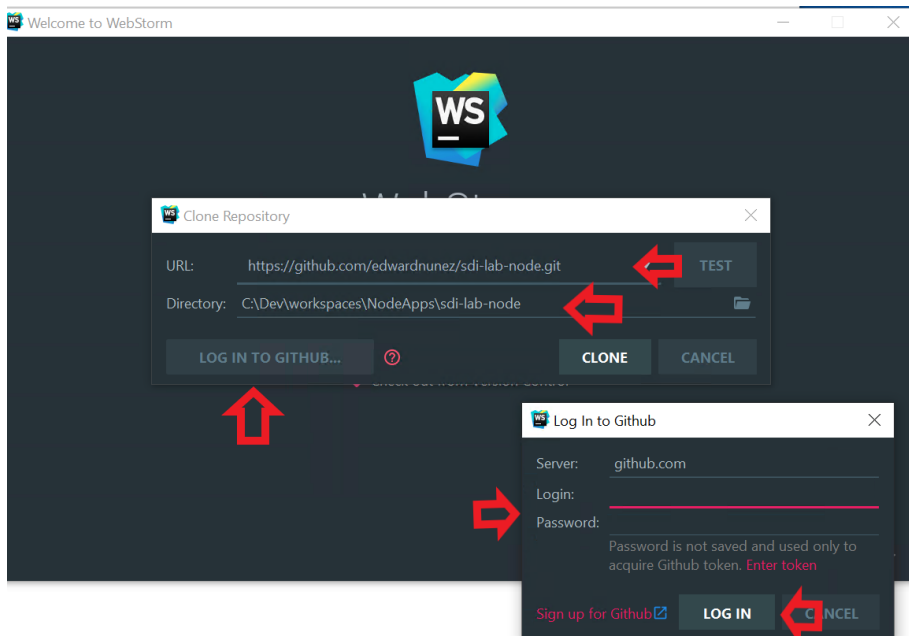
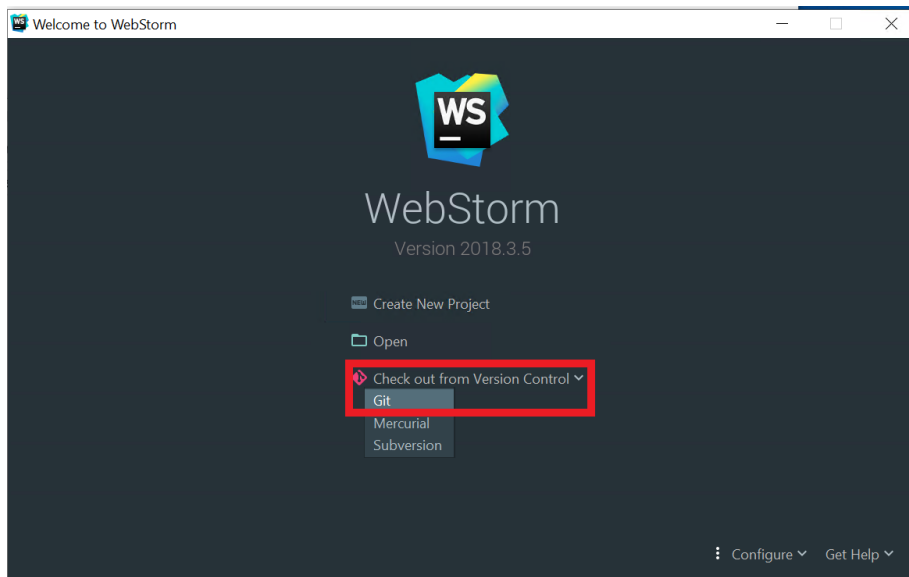
Add .gitignore: Node | Add a license: None

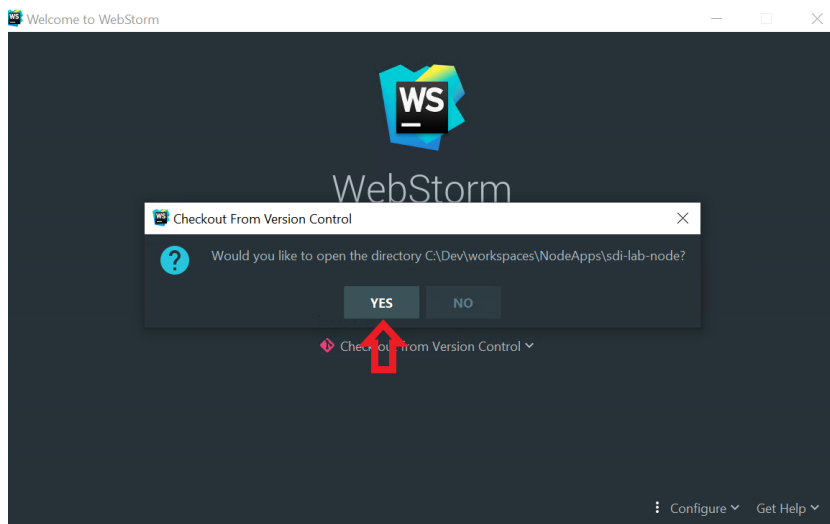
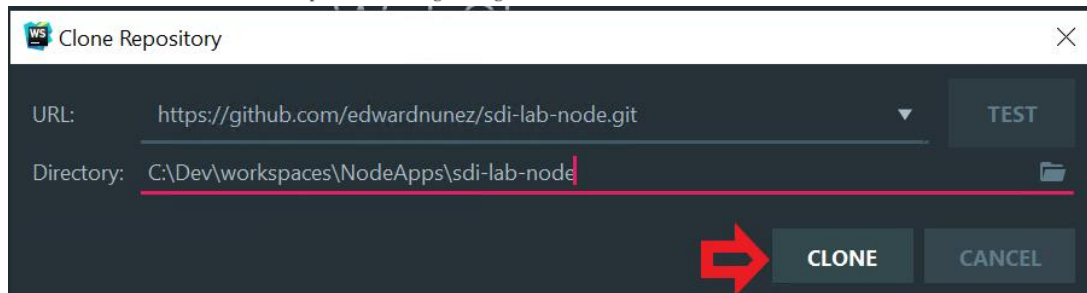
Create repository



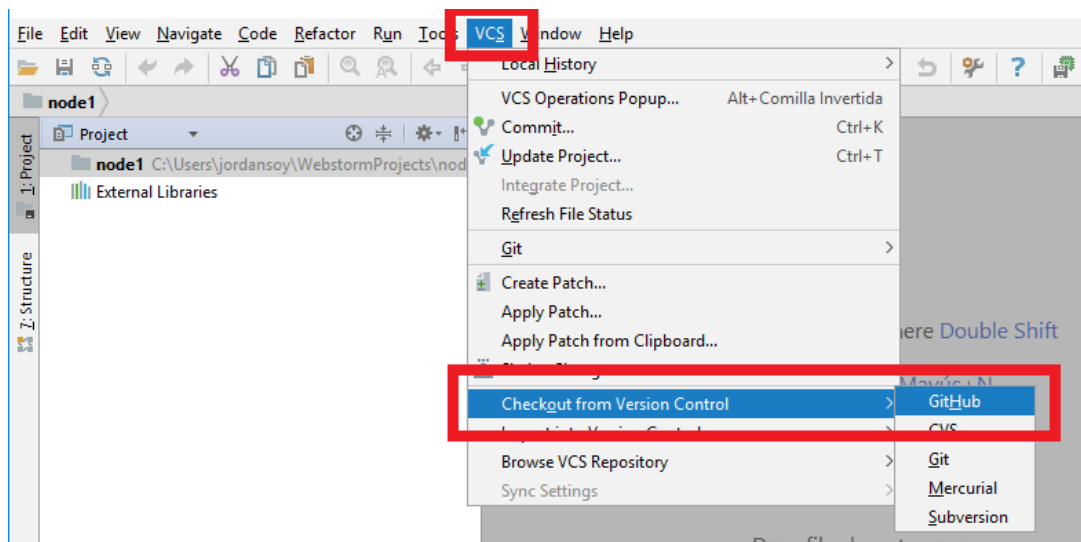
2 Creación del proyecto

Vamos a crear un proyecto NodeJS desde cero, aunque lo normal es siempre partir de una plantilla de la que nos ofrecen los IDE para agilizar el desarrollo. Nuestro proyecto se llamará **sdi-lab-node**, y lo vamos a sincronizar el repositorio de GIT creado anteriormente. Si es la primera vez que iniciamos el IDE, nos aparecerá lo haremos desde la pantalla de presentación:



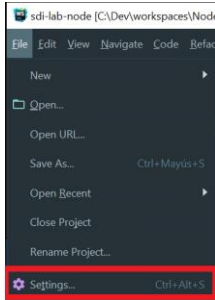


Si no aparece la ventana de presentación, accedemos a esta misma opción desde VCS → Checkout from Version Control > Github. Desde esa opción será posible iniciar sesión en GitHub y descargar cualquier proyecto.

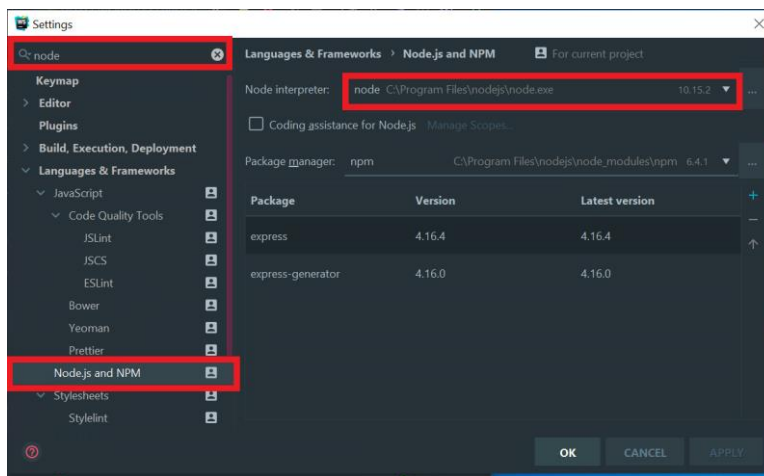




Lo primero que vamos a realizar es confirmar que el IDE apunta a la versión de Node correspondiente, para esto vamos al menú **File -> Settings...**

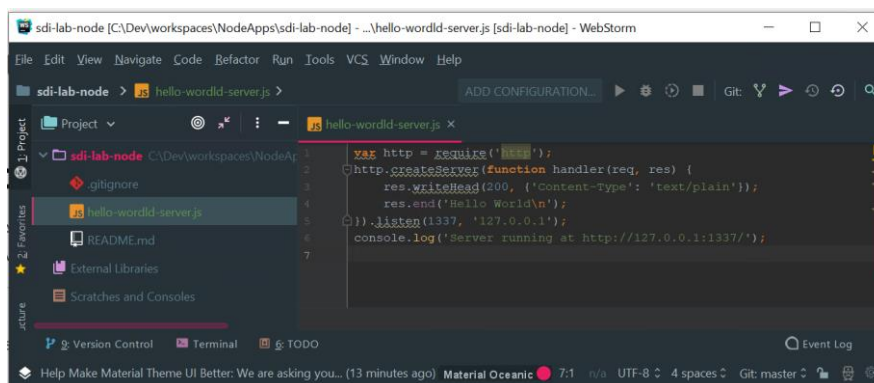


Buscamos **Node** en el cuadro de texto y comprobamos que la ruta es correcta:



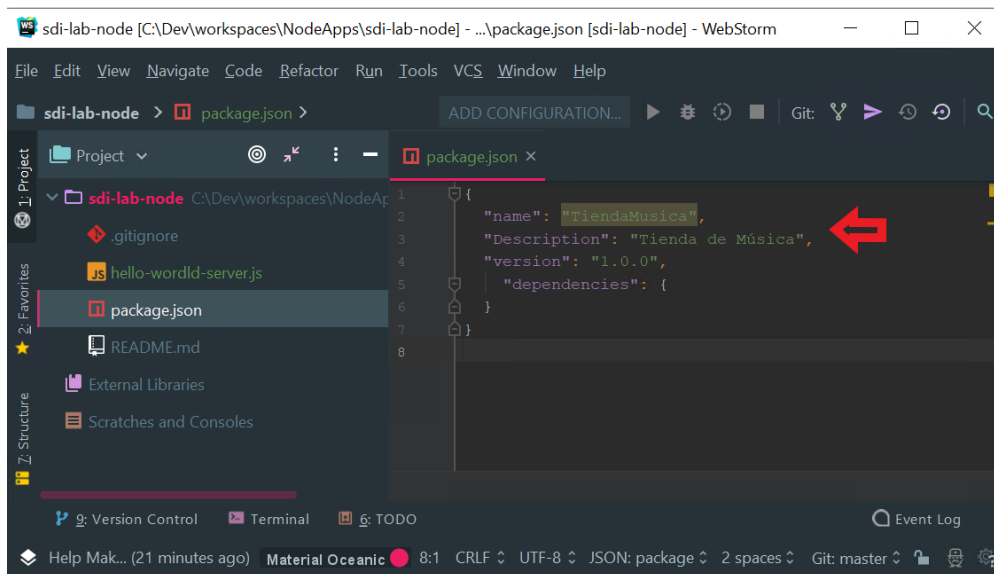
A continuación, vamos a crear un ejemplo estilo “Hola Mundo”. Creamos un fichero **File -> New -> JavaScript File** y lo llamamos **hello-world-server.js** que contendrá la lógica de la aplicación. El código que añadiremos definirá una pequeña aplicación web Node usando un módulo muy básico: **http**. Para ello, añadimos el siguiente código:

```
var http = require('http');
http.createServer(function handler(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

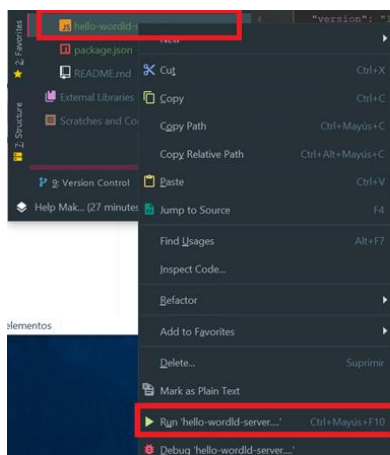




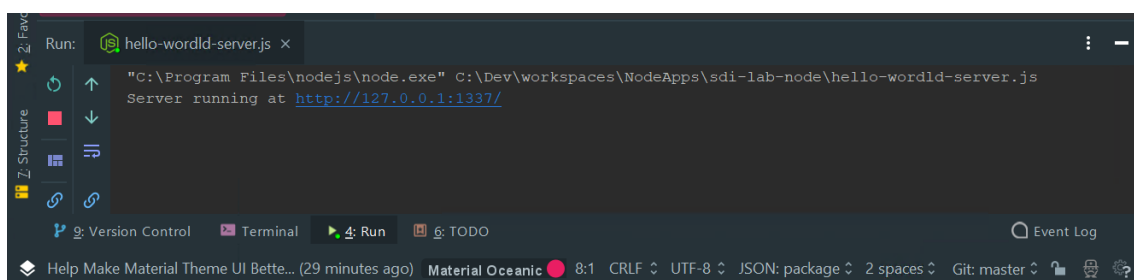
Antes de ejecutar el proyecto, creamos el fichero **package.json** para definir la configuración y los metadatos de la aplicación. Vamos al menú **File → New → Package.json** y cambiamos el nombre de la aplicación por “tienda_musica” y añadimos como descripción “Tienda de Música”.



Para ejecutar la aplicación, hacemos click derecho en el fichero **hello-world-server.js** y seleccionamos: **Run 'hello-word-server...'**.

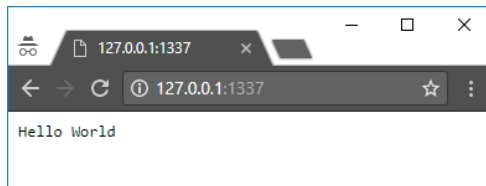


En la consola podremos ver el estado del despliegue y los mensajes impresos por el console.log

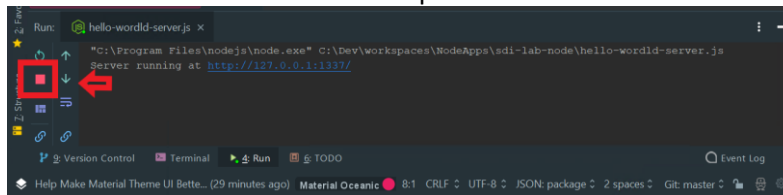




Desde <http://127.0.0.1:1337/> podemos acceder a nuestra aplicación.

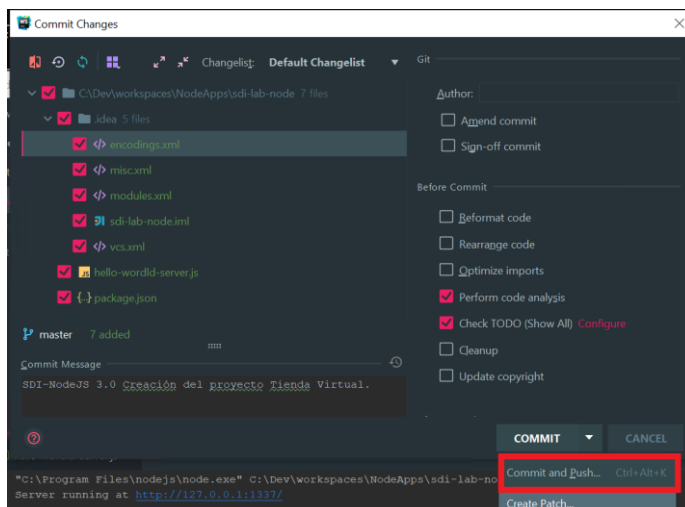


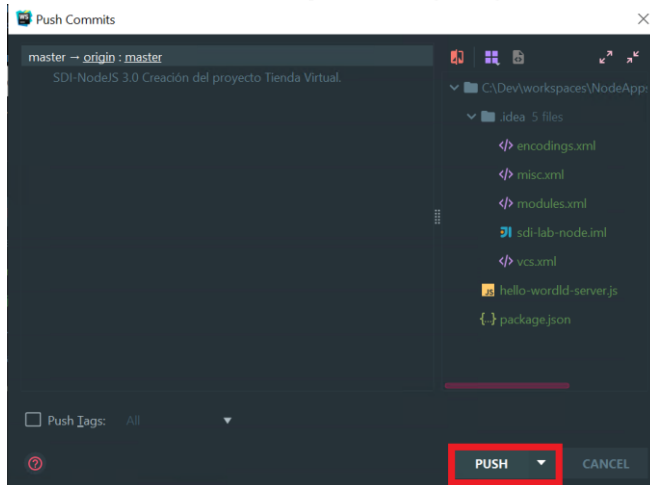
Para detener el servidor debemos pulsar en el botón de **detener** o las teclas **Ctrl+F2**



Nota: Subir el código a GitHub en este punto. Commit Message → *“SDI-NodeJS-0 Creación del proyecto Tienda Virtual”*.

Click derecho sobre el proyecto → **Git** → **Commit Directory**





3 Express

Express es un framework de desarrollo de aplicaciones web minimalista y flexible para NodeJS. A lo largo de los próximos guiones, utilizaremos este framework para crear una aplicación web completa.

Creamos un **nuevo fichero app.js**, es el nombre más extendido para el fichero principal de la aplicación web. (**New** → **File**, Filename: **app.js**).

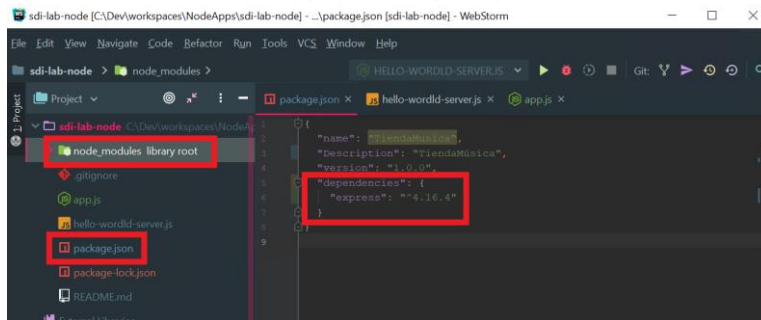
El módulo Express no es nativo de NodeJS, por lo que tenemos que instalarlo a través del sistema de gestión de paquetes NPM. Abrimos la consola de línea comandos CMD y **nos situamos en el directorio raíz del proyecto**. Ejecutamos el comando **npm install express --save**. Añadir el parámetro **--save** hace que express se añada a nuestra lista de dependencias del proyecto (según la versión puede que no sea necesario).

```
C:\Dev\workspaces\NodeApps\sdi-lab-node>npm install express --save
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN TiendaMusica@1.0.0 No description
npm WARN TiendaMusica@1.0.0 No repository field.
npm WARN TiendaMusica@1.0.0 No license field.

+ express@4.16.4
added 48 packages from 36 contributors and audited 121 packages in 4.11s
found 0 vulnerabilities
```

Cuando instalamos un nuevo módulo se producen cambios en el proyecto (quizá haya que actualizar/recargar el proyecto en el IDE para ver los cambios).

- Se añade una dependencia automáticamente en el fichero **package.json**.
- Se añade una nueva carpeta **node_modules**, con el código de los módulos.



Incluimos el siguiente código en el fichero **app.js**.

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

app.get('/usuarios', function(req, res) {
  console.log("Depurar aquí");
  res.send('ver usuarios');
});

app.get('/canciones', function(req, res) {
  res.send('ver canciones');
});

// lanzar el servidor
app.listen(8081, function() {
  console.log("Servidor activo");
});
```

La función **require** indica que se utilizará un determinado módulo. En este caso, hacemos uso del módulo “express” para almacenarlo en la variable **express**. A continuación, mediante la función **express()** crearemos **una aplicación que almacenaremos en la variable app**.

Podemos habilitar que la aplicación responda a peticiones **get** utilizando **app.get(<ruta>,<función de respuesta>)**. La función de respuesta recibe dos parámetros **req** (datos de la request/petición) **res** (datos de la response/respuesta). Para que la aplicación responda debemos introducir datos en la respuesta **res**, utilizamos **res.send()** para retornar texto como respuesta.

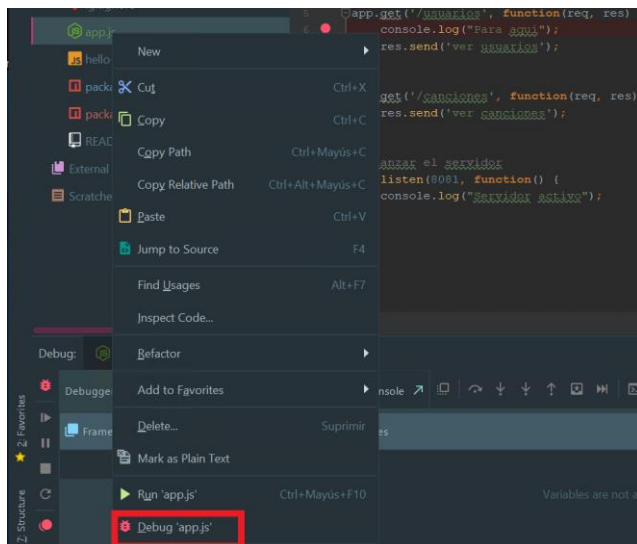
4 Depurar

Depurar utilizando un IDE como WebStorm es muy sencillo. Para ello, **colocamos uno o varios puntos de ruptura haciendo click junto al número de la línea a depurar**.



```
1 // Módulos
2 var express = require('express');
3 var app = express();
4
5 app.get('/usuarios', function(req, res) {
6   console.log("Para aquí");
7   res.send('ver usuarios');
8 });
9
10 app.get('/canciones', function(req, res) {
11   res.send('ver canciones');
12 });
13
14 // lanzar el servidor
15 app.listen(8081, function() {
16   console.log("Servidor activo");
17 });
```

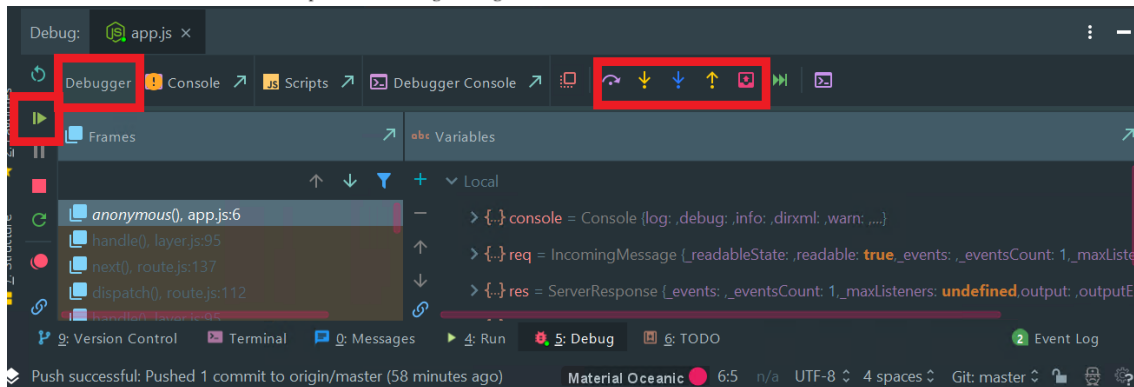
Para ejecutar en modo depuración pulsamos botón derecho sobre el fichero principal, **app.js** y seleccionamos **Debug 'app.js'** los botones de depuración del IDE para: pasar al siguiente punto de ruptura, detener la ejecución o avanzar paso a paso.



Si tenemos el puerto libre la aplicación se ejecutará el **modo de depuración**.

Al realizar una petición cuando se encuentre un punto de ruptura, se detendrá la ejecución. Por ejemplo, si accedemos desde el navegador a la URL <http://localhost:8081/usuarios>, en la pestaña **Debugger** podremos ver toda la información de depuración:

- Las flechas de la parte central derecha nos permiten hacer avanzar la ejecución.
- La flecha verde de la parte izquierda reanuda la ejecución hasta el siguiente punto de ruptura.



5 Variables de entorno

En la variable que utilizamos para almacenar la aplicación (app en nuestro caso), es posible **almacenar variables de entorno utilizando los métodos app.get y app.set**. Vamos a utilizar este sistema para almacenar una variable de configuración: el puerto en el que se despliega la aplicación.

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

// Variables
app.set('port', 8081);

app.get('/usuarios', function(req, res) {
  res.send('ver usuarios');
})

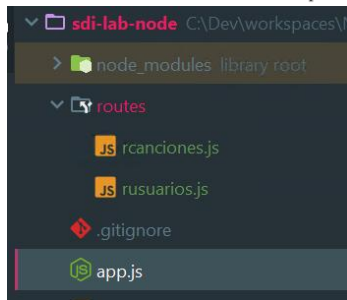
app.get('/canciones', function(req, res) {
  res.send('ver canciones');
})

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

Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS-.3,4,5 Express, depuración y variables de entorno".*

6 División de las rutas en módulos

Creemos la carpeta **/routes** y dentro de ella los ficheros **rusuarios.js** y **rcanciones.js**. Cada uno de ellos se encargará de gestionar las rutas de una entidad (usuarios y canciones, respectivamente). En cada fichero, **crearemos una función que exportaremos como módulo** utilizando **module.exports**. Así, cada módulo podrá recibir parámetros (gracias a la función) cuando hagamos require. En este caso recibimos una referencia a **app**.



Contenido de `/routes/rcanciones.js`

```
module.exports = function(app) {  
    app.get("/canciones", function(req, res) {  
        res.send("ver canciones");  
    });  
};
```

Contenido de `/routes/rusuarios.js`

```
module.exports = function(app) {  
    app.get("/usuarios", function(req, res) {  
        res.send("ver usuarios");  
    });  
};
```

Finalmente eliminaremos las respuestas get de **app.js**, sustituyéndolas por el require de los módulos **rusuarios** y **rcanciones**. Además, debemos enviar la variable **app** como parámetro.

```
// Módulos  
var express = require('express');  
var app = express();  
  
// Variables  
app.set('port', 8081);  
  
app.get('/usuarios', function(req, res) {  
    res.send('ver usuarios');  
});  
  
app.get('/canciones', function(req, res) {  
    res.send('ver canciones');  
});  
  
//Rutas/controladores por lógica  
require("./routes/rusuarios.js")(app); // (app, param1, param2, etc.)  
require("./routes/rcanciones.js")(app); // (app, param1, param2, etc.)  
  
// lanzar el servidor  
app.listen(app.get('port'), function() {  
    console.log("Servidor activo");  
})
```



7 Peticiones GET y parámetros

Las peticiones GET pueden contener parámetros en su URL. Existen dos formas comunes de enviar parámetros en una petición (request / req). **Una de ellas es concatenando el nombre del parámetro(s) y su valor(es) al final de la URL**, como puede verse en los siguientes ejemplos:

- <http://localhost:8081/canciones?nombre=despacito>
Parámetro con clave “nombre” y valor “despacito”, se agrega con el operador ?
- <http://localhost:8081/canciones?nombre=despacito&autor=Luis Fonsi>
Igual que el ejemplo anterior, pero con un parámetro 2 con clave “autor” y valor “Luis Fonsi”. Todos los parámetros a partir del primero se concatenan con el operador &

Para obtener los parámetros enviados en una petición GET con la sintaxis anterior, hacemos uso de **req.query.<clave_parámetro>**. Por ejemplo, para obtener el parámetro **nombre** y **autor** de la petición **/GET canciones**, deberíamos hacer lo siguiente:

```
module.exports = function(app) {  
  app.get("/canciones", function(req, res) {  
    let respuesta = 'Nombre: ' + req.query.nombre + '<br>' + 'Autor: ' + req.query.autor;  
    res.send(respuesta);  
  });  
};
```

Probamos a ejecutar las siguientes URLs:

- <http://localhost:8081/canciones?nombre=despacito>
- <http://localhost:8081/canciones?nombre=despacito&autor=Luis Fonsi>

En la primera URL podemos observar que los parámetros son opcionales. En el caso de no encontrar el parámetro solicitado con **req.query.<clave_del_parámetro>**, obtendremos como retorno **“undefined”**.

Para comprobar que la variable tiene valor podemos comprobar si el valor es distinto de null, o si el tipo **typeof()** es distinto de “undefined”.

```
module.exports = function(app) {  
  app.get("/canciones", function(req, res) {  
    let respuesta = "";  
    if (req.query.nombre != null)  
      respuesta += 'Nombre: ' + req.query.nombre + '<br>';  
    if (typeof (req.query.autor) != "undefined")  
      respuesta += 'Autor: ' + req.query.autor;  
    res.send(respuesta);  
  });  
};
```



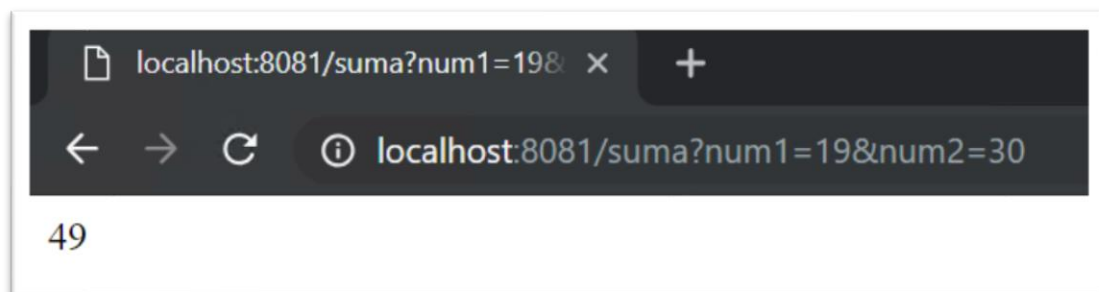
Todos los valores que obtenemos a través del **req.query** son cadenas de texto. Por tanto, si quisiéramos tratarlas como enteros habría que convertirlas primero. En el siguiente ejemplo se **intenta** sumar dos números que son enviados como parámetros.

Añade el siguiente método en el módulo `rcanciones`.

```
module.exports = function(app) {  
  ...  
  app.get('/suma', function(req, res) {  
    let respuesta = req.query.num1 + req.query.num2;  
    res.send(respuesta);  
  });  
};
```

Si accedemos a <http://localhost:8081/suma?num1=19&num2=30> vemos que realmente **no se están sumando, se están concatenando ya que son cadenas**. Así pues, habría que pasarlos a enteros empleando la función `parseInt()`, operar con ellos y convertirlos a cadena para enviarlos con `res.send`, que solamente admite cadenas.

```
app.get('/suma', function(req, res) {  
  let respuesta = parseInt(req.query.num1) + parseInt(req.query.num2);  
  res.send(String(respuesta));  
});
```



Otra forma de enviar parámetros GET es inyectándolos en la URL entre barras: `/<valor_parámetro>/`. En este caso, **no se especifica la clave que tiene**, puesto que es en la implementación del controlador la que determina la clave del parámetro **según la posición de éste en la URL**. Este sistema se suele emplear para ids y categorización:

- <http://localhost:8081/canciones/121/>
Canción con parámetro "id" = 121
- <http://localhost:8081/canciones/pop/121/>
Canción con parámetro "genero" = pop e "id" = 121

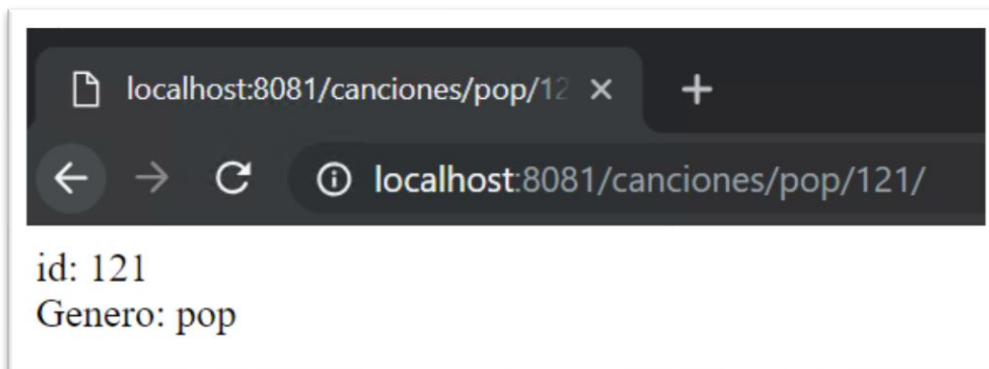
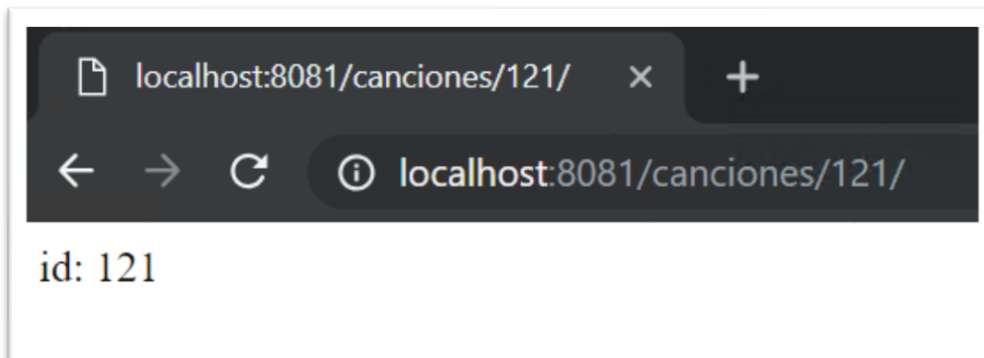
Estos parámetros se corresponderían con estas claves si la aplicación así lo especificase, los parámetros se deben especificar en la ruta con `:<clave_del_parámetro>` y se accede a ellos en



el código utilizando **req.params.<clave_del_parámetro>**. Añade los siguientes métodos en el módulo **rcanciones**.

```
app.get('/canciones/:id', function(req, res) {  
  let respuesta = 'id: ' + req.params.id;  
  res.send(respuesta);  
});  
  
app.get('/canciones/:genero/:id', function(req, res) {  
  let respuesta = 'id: ' + req.params.id + '<br>' +  
    'Género: ' + req.params.genero;  
  
  res.send(respuesta);  
});
```

Si probamos las URL anteriores en el navegador, deberíamos obtener como resultados:



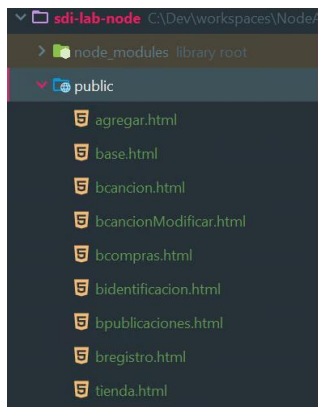
Nota: Subir el código a GitHub en este punto. Commit Message → *“SDI-NodeJS- 6,7 División de las rutas en módulos, peticiones GET y parámetros”.*



8 Recursos estáticos

Express provee una función de asistencia (middleware) para facilitar el acceso de los clientes a ficheros estáticos: HTMLs estáticos, imágenes, videos, css, etcétera. Para ello, es necesario crear un directorio (por convenio se le suele llamar **public**) y declararlo en la aplicación mediante la expresión **express.static('<nombre del directorio>')**.

Creamos la carpeta public en la raíz del **proyecto y descomprimos dentro el CONTENIDO (incluida la carpeta img) de la carpeta recursos disponible en PL-SDI-Material7.zip** (está en el Campus Virtual). A continuación, se muestra la estructura del directorio tras el proceso:



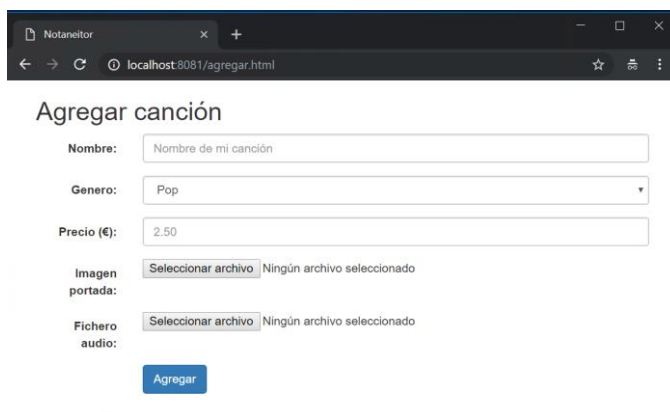
El contenido de las carpetas estáticas se puede modificar sin falta de reiniciar la aplicación. Seguidamente, declaramos la ruta **public** como estática en **app.js**

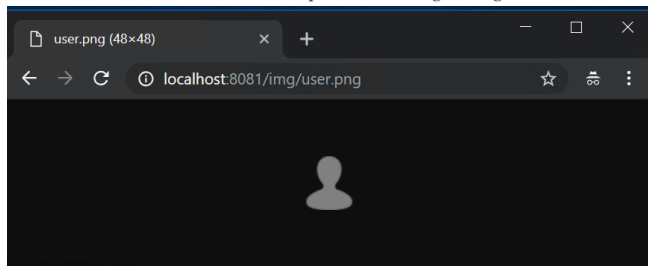
```
// Módulos
var express = require('express');
var app = express();

app.use(express.static('public'));
```

Ejecutamos la aplicación y comprobamos que podemos acceder a los recursos de esta carpeta:

- <http://localhost:8081/agregar.html>
- <http://localhost:8081/img/user.png>





9 Peticiones POST y parámetros

A diferencia de las peticiones GET, **las peticiones POST tienen un cuerpo (body) que puede contener datos** (pares de clave-valor, texto plano, json, binario, o cualquier otro tipo de datos). Este tipo de peticiones se utilizan comúnmente para enviar formularios.

El fichero localizado en la ruta <http://localhost:8081/agregar.html> define un formulario que envía una petición **POST /cancion**. Este formulario contiene varios inputs con atributo **name** (que funcionan como clave): nombre, genero, precio, portada y audio. En el siguiente ejemplo, se muestra un fragmento de **agregar.html** donde se define la clave (o name) para el campo precio.

```
<div class="col-sm-10">  
  <input type="number" class="form-control" name="precio" placeholder="2.50" required="true" />
```

*https://www.w3schools.com/tags/tag_form.asp

Vamos a hacer que nuestra aplicación procese la petición **POST /cancion**. Para poder acceder a los parámetros que se envían a través del body, necesitamos añadir el módulo externo **body-parser** <https://www.npmjs.com/package/body-parser-json>. Desde el CMD debemos acceder al directorio raíz de nuestro proyecto y ejecutamos el comando de instalación del módulo body-parser: **npm install body-parser --save**

```
c:\Dev\workspaces\NodeApps\sdi-lab-node>npm install body-parser --save  
npm WARN tiendamusica@1.0.0 No description  
npm WARN tiendamusica@1.0.0 No repository field.  
npm WARN tiendamusica@1.0.0 No license field.  
  
+ body-parser@1.18.3  
updated 1 package and audited 151 packages in 2.239s  
found 0 vulnerabilities
```



En el fichero principal **app.js**, nos aseguramos de añadir el require del módulo **"body-parse"**. En las siguientes líneas, declaramos el uso de las funciones parseadoras mediante **app.use**:

- **bodyParser.json()**: para poder procesar JSON.
- **bodyParser.urlencoded()**: para poder procesar formularios estándar.



El código a incluir en el app.js es el siguiente:

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

let bodyParser = require('body-parser');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.use(express.static('public'));
```

Al haber añadido el módulo body-parser, podemos acceder al contenido de las peticiones POST empleando: **req.body.<nombre_del_parámetro>**. Abrimos el fichero **rcanciones.js** y creamos una función **app.post** que procese los atributos nombre, género y precio enviados a través del formulario.

```
app.post("/cancion", function(req, res) {
  res.send("Canción agregada:" + req.body.nombre + "<br>"
    + " género : " + req.body.genero + "<br>"
    + " precio: " + req.body.precio);
});
```

Guardamos los cambios, ejecutamos la aplicación, utilizamos el formulario <http://localhost:8081/agregar.html> y comprobamos que funciona correcta. **¡Ojo! La aplicación aún no agrega los ficheros y lo resolveremos en apartados posteriores.**

← → ↻ ⓘ localhost:8081/agregar.html

Agregar canción

Nombre:

Genero:

Precio (€):

Imagen portada:
 Ningún archivo seleccionado

Fichero audio:
 Ningún archivo seleccionado

← → ↻ ⓘ localhost:8081/cancion

Canción agregada:nuev
genero :pop
precio: 2



10 Enrutamiento y comodines

Dentro de la especificación de rutas, se admite el uso de comodines (`?`, `+`, `*` y `()`) y otras expresiones regulares. Por ejemplo, la siguiente ruta responderá a cualquier petición que empiece por “**promo**”: `/promo`, `/promocion`, `/promocionar` y similares.

```
app.get('/promo*', function (req, res) {  
  res.send('Respuesta patrón promo* ');  
})
```

Lista de ejemplos de enrutamiento con comodines y expresiones regulares:

<http://expressjs.com/es/guide/routing.html>.

Nota: Subir el código a GitHub en este punto. Commit Message → “SDI-NodeJS- 8,9,10 Recursos estáticos, Peticiones POST y parámetros, Enrutamiento y comodines”.

11 Vistas y Motores de plantillas

Una de las formas más comunes para intercalar datos de la lógica de negocio en los ficheros de presentación HTML se basa en el uso de motores de plantillas. Éstos nos permiten combinar texto de salida (por ejemplo HTML) y datos procedentes de la lógica, aplicando diferentes etiquetas, funciones y filtros propios del motor.

Podemos combinar el framework Express con muchos motores de plantillas. **En prácticas vamos a utilizar Swig**, un motor con un buen nivel de funcionalidad y con bastantes similitudes a **twig** (muy popular en otras tecnologías) <http://node-swig.github.io/swig-templates/>

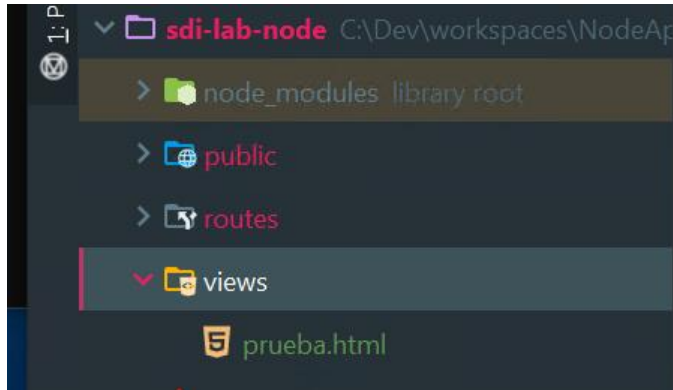
Para instalar el módulo de **swig** en nuestro proyecto abrimos la línea de comandos CMD y, en el directorio raíz del proyecto, ejecutamos el comando **npm install swig@1.4.2 --save**

```
c:\Dev\workspaces\NodeApps\sdi-lab-node>npm uninstall swig@1.4.2 --save  
npm WARN tiendamusica@1.0.0 No description  
npm WARN tiendamusica@1.0.0 No repository field.  
npm WARN tiendamusica@1.0.0 No license field.  
  
audited 165 packages in 1.296s  
found 1 low severity vulnerability  
  run `npm audit fix` to fix them, or `npm audit` for details
```



Como no es nada recomendable almacenar vistas en `/public`, vamos a crear una nueva carpeta `/views` en el directorio raíz. Dentro de ésta, creamos un fichero `prueba.html`.

Nota: En Swig, las plantillas pueden tener cualquier extensión incluyendo `.html`, no tienen una extensión específica como en otros entornos.



La plantilla va a contener tanto el código HTML como los atributos enviados desde el controlador a la vista. En este caso, los atributos serán:

- **vendedor:** parámetro de tipo cadena con el nombre del vendedor.
- **canciones:** lista de objetos de tipo canción, donde cada canción tendrá un atributo **nombre** y **precio**.

Para acceder al valor de los atributos utilizamos:

`{{ <nombre atributo> }}`

Para recorrer una lista de atributos utilizamos:

`{ % for elemento in <nombre_lista> % } ... { % endfor % }`

El código a añadir a la plantilla `prueba.html`, sería el siguiente:

```
<html>
  <head>
    <title>Canciones</title>
  </head>
  <body>
    <h1>{{ vendedor }}</h1>
    <ul>
      {% for cancion in canciones %}
        <li>
          {{ cancion.nombre }} - {{ cancion.precio }}
        </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Se pueden incluir muchas otras etiquetas de procesamiento: *else if*, *for*, *block*... Podéis ampliar información en: <http://node-swig.github.io/swig-templates/docs/tags/>



Ahora vamos a incluir el motor de plantillas en la aplicación, importando el módulo mediante **require('swig')**. A través de la variable **swig**, seremos capaces de renderizar plantillas.

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

let swig = require('swig');
let bodyParser = require('body-parser');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static('public'));
```

Con el fin de que los controladores utilicen Swig, debemos pasar la variable **swig**, como parámetro a los módulos que actúan como controladores.

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

Y, por tanto, debemos incluir el parámetro en ambos módulos: **rcanciones** y **usuarios**.

```
module.exports = function(app, swig) {
  app.get("/canciones", function(req, res) {
    res.send("Ver canciones");
  });

  app.post("/cancion", function(req, res) {
    res.send("Canción agregada:" + req.body.nombre + "<br>"
      + " genero : " + req.body.genero + "<br>"
      + " precio: " + req.body.precio);
  });
};
```

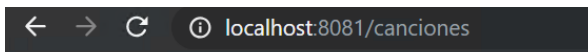
```
module.exports = function(app, swig) {
  app.get("/usuarios", function(req, res) {
    res.send("ver usuarios");
  });
};
```



Cuando se reciba una petición **GET /canciones**, crearemos un array con 3 canciones, cada una con nombre y precio. La función **swig.renderFile** recibe como parámetros: la ruta de la plantilla y los parámetros que podrá usar ésta(string **vendedor** y array de **canciones**). El retorno de la función será el HTML final procesado por la plantilla y será enviado como respuesta (**res.send**).

```
app.get("/canciones", function(req, res) {  
    let canciones = [ {  
        "nombre": "Blank space",  
        "precio": "1.2"  
    }, {  
        "nombre": "See you again",  
        "precio": "1.3"  
    }, {  
        "nombre": "Uptown Funk",  
        "precio": "1.1"  
    } ];  
  
    let respuesta = swig.renderFile('views/prueba.html', {  
        vendedor: 'Tienda de canciones',  
        canciones: canciones  
    });  
  
    res.send(respuesta);  
});
```

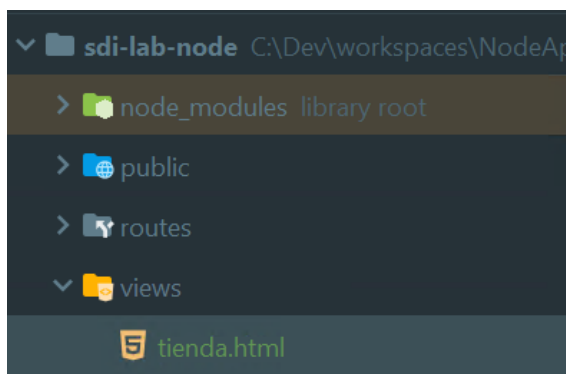
Ejecutamos la aplicación y accedemos a <http://localhost:8081/canciones>. Deberíamos ver la plantilla correctamente procesada.



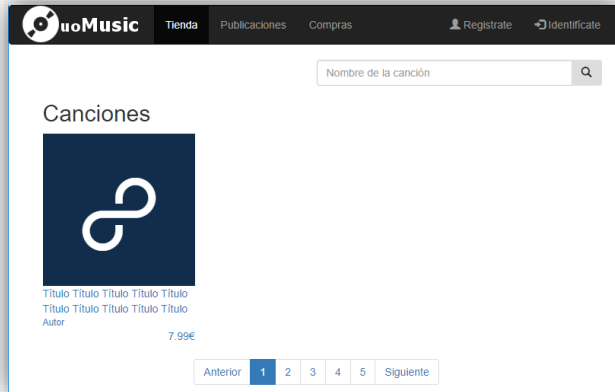
Tienda de canciones

- Blank space - 1.2
- See you again - 1.3
- Uptown Funk - 1.1

A continuación, cambiaremos la plantilla **prueba.html** por otra que incluya un interfaz de usuario más completa. Movemos el fichero **tienda.html** (actualmente en **/public**) a la carpeta **/views**.



El fichero tienda.html incluye el catálogo de una tienda y utiliza el framework **Bootstrap 3** <https://getbootstrap.com/docs/3.3/> . Si analizamos el fichero veremos que, por ahora, solo incluye código HTML y ninguna funcionalidad de Swig.



Abrimos el fichero **tienda.html** y localizamos el bloque donde se introduce la información de la canción.

```
<h2>Canciones</h2>
<div class="row">

  <!-- http://librosweb.es/libro/bootstrap_3/capitulo_2/tipos_de_rejillas.html -->
  <!-- Inicio del Bloque canción -->
  <div class="col-xs-6 col-sm-6 col-md-4 col-lg-3">
    <div style="width:200px">
      <a href="/cancion/id">
        
        <!-- http://www.socicon.com/generator.php -->
        <div class="wrap">Título Título Título Título Título Título Título
        <div class="small">Autor</div>
        <div class="text-right">7.99€</div>
      </a>
    </div>
  </div>
  <!-- Fin del Bloque canción -->
</div>
```

Sustituimos el código HTML por el script que recorre la lista de **canciones** y muestra su información. Aún no tenemos toda la información que queremos mostrar (véase el autor), pero la incluiremos igualmente.

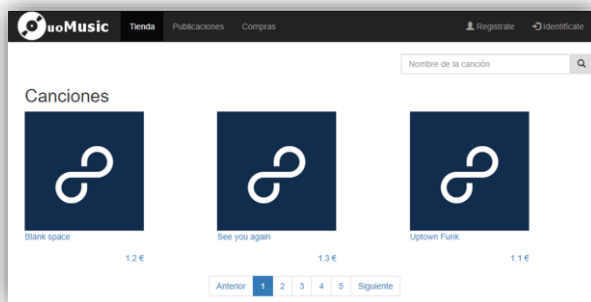
```
<!-- Inicio del Bloque canción -->
{% for cancion in canciones %}
<div class="col-xs-6 col-sm-6 col-md-4 col-lg-3">
  <div style="width:200px">
    <a href="/cancion/id">
      
      <!-- http://www.socicon.com/generator.php -->
      <div>{{ cancion.nombre }}</div>
      <div class="small">{{ cancion.autor }}</div>
      <div class="text-right">{{ cancion.precio }} €</div>
    </a>
  </div>
</div>
{% endfor %}
<!-- Fin del Bloque canción -->
```



En el fichero **rcanciones.js**, modificamos la función **GET /canciones** para **que procese el fichero tienda.html** en lugar del fichero prueba.html

```
let respuesta = swig.renderFile('views/tienda.html', {  
  vendedor : 'Tienda de canciones',  
  canciones : canciones  
});
```

Accedemos a la página y comprobamos que el resultado es correcto.



12 Plantillas – URLs absolutas

Es muy común que las plantillas accedan a recursos estáticos: css, imágenes, fuentes, ficheros js, etcétera. Estos recursos se almacenan en un **directorio estático** (para eso hemos creado previamente el directorio **public**). En nuestro caso **tienda.html** accede a dos recursos estáticos:

El icono de la barra de navegación

```

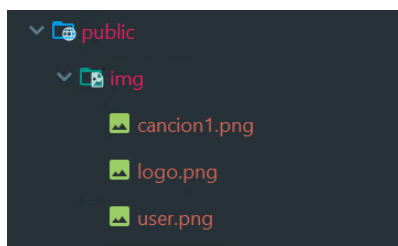
```

Las caratulas de las canciones:

```

```

Al cargar localhost:8081/canciones, el navegador busca estas dos imágenes en localhost:8081/img/logo.png y localhost:8082/img/cancion1.png. **Como en el apartado 8 definimos public como carpeta estática**, el servidor busca y envía con éxito las imágenes desde public/img/logo.png y public/img/cancion1.png.





Sin embargo, ¿Qué pasaría si la URL base fuera <http://localhost:8081/nuevas/canciones?>

```
module.exports = function(app, swig) {  
  app.get("/nuevas/canciones", function(req, res) {  
    var canciones = [ {  
      "nombre" : "Blank space",
```

Al especificar las imágenes **img/logo.png** e **img/cancion1.png** de forma relativa, la aplicación buscaría los recursos en **public/nuevas/img/logo.png** y **public/nuevas/img/cancion1.png**. **Obviamente, el directorio nuevas no existe.** Una buena práctica para prevenir problemas de este tipo suele ser usar rutas absolutas. Modificamos el documento para introducir rutas absolutas.

```

```

```

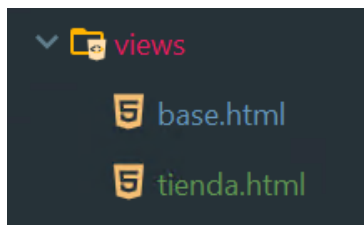
```

Nota: los cambios en las plantillas no se reconocen de forma dinámica, hay que volver a desplegar la aplicación.

13 Plantillas – Bloques

Cuando tenemos una aplicación web con varias vistas lo más frecuente suele ser que gran parte de la interfaz sea común en todas ellas. A nivel de programación, con el fin de tener código más mantenible y reutilizable, utilizamos una plantilla base con bloques identificados que, en cada caso, serán sustituidos por otros contenidos.

Copiamos en la carpeta **/view** el fichero **base.html** que se encuentra actualmente en **/public/**. Básicamente es una página sin contenido pero que marca la estructura del sitio web.



Dentro de **base.html** vamos a identificar los bloques donde se deberían agregar contenidos utilizando las etiquetas **{% block <id_del_bloque %} {% endblock %}**. Siendo posible identificar tantos bloques como queramos dentro de la página, nos interesara incluir bloques en todas las zonas cuyo contenido varíe según el contexto de la aplicación.



Vamos a comenzar incluyendo un bloque para el contenido principal, con la id **contenido_principal**. Este bloque lo colocaremos dentro del div **class=container**.

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

Definimos también un bloque al inicio del fichero **base.html** para poder cambiar el título de la página; lo llamamos **titulo**.

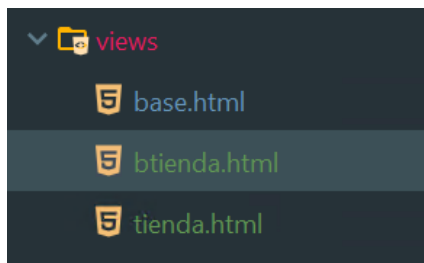
```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>{% block titulo %} uoMusic {% endblock %}</title>
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1"/>
</head>
<body>
```

Suele ser de mucha utilidad incluir un bloque en la cabecera para incluir de forma variable ficheros CSS y JavaScript. Llamaremos a este bloque **scripts**.

```
<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>
{% block scripts %} {% endblock %}
</head>
<body>
```

Por el momento disponemos de los bloques: **contenido_principal**, **titulo** y **scripts**. Estos bloques podrán ser redefinidos (o no) en todas las plantillas que hereden de **base.html**

Vamos a crear un nuevo fichero **btienda.html** en la carpeta **/views/**. Este fichero extenderá de la plantilla **base.html** y definirá el contenido de parte de los bloques anteriores.



Abrimos el fichero **btienda.html** y agregamos la directiva **{% extends "base.html" %}**. A partir de aquí, incluimos solamente la redefinición de los bloques que nos interesen, con la sintaxis: **{% block <nombre_del_bloque> %} contenido redefinido {% endblock %}**.



Definimos el contenido para los bloques **título** y **contenido principal**. Así pues, el contenido de **btienda.html** será el siguiente:

```
{% extends "base.html" %}

{% block titulo %} Tienda - uoMusic {% endblock %}

{% block contenido_principal %}
<h2>Canciones</h2>
<div class="row">

    <!-- http://librosweb.es/libro/bootstrap_3/capitulo_2/tipos_de_rejillas.html -->
    <!-- Inicio del Bloque canción -->
    {% for cancion in canciones %}
    <div class="col-xs-6 col-sm-6 col-md-4 col-lg-3">
        <div style="width: 200px">
            <a href="/cancion/id"> 
                <!-- http://www.socicon.com/generator.php -->
                <div class="wrap">{{ cancion.nombre }}</div>
                <div class="small">Autor</div>
                <div class="text-right">{{ cancion.precio }} €</div>
            </a>
        </div>
    </div>
    {% endfor %}
    <!-- Fin del Bloque canción -->
</div>
{% endblock %}
```

Cambiamos la respuesta de **GET /canciones** para que retorne la nueva vista **btienda.html**

```
app.get("/canciones", function(req, res) {

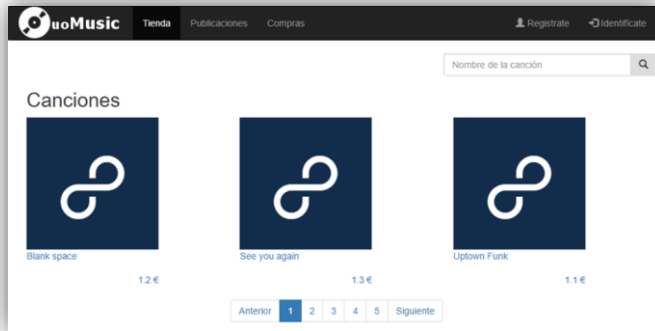
    var canciones = [ {
        "nombre" : "Blank space",
        "precio" : "1.2"
    }, {
        "nombre" : "See you again",
        "precio" : "1.3"
    }, {
        "nombre" : "Uptown Funk",
        "precio" : "1.1"
    } ];

    var respuesta = swig.renderFile('views/btienda.html', {
        vendedor : 'Tienda de canciones',
        canciones : canciones
    });

    res.send(respuesta);

});
```

Guardamos los cambios y ejecutamos la aplicación para comprobar el funcionamiento.



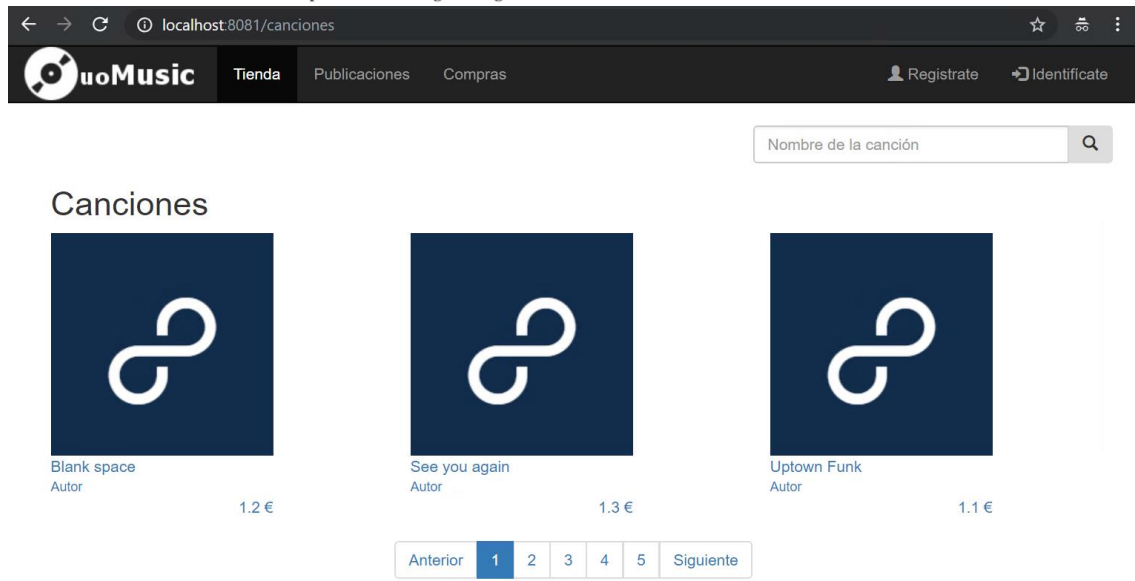
En el fichero **btienda.html**, vamos a añadir el código HTML para un sistema de búsqueda y uno de paginación. Por ahora, no implementaremos la funcionalidad de estos sistemas, simplemente tendremos preparada la vista.

```
{% block contenido_principal %}
<!-- Búsqueda -->
<div class="row">
  <div id="custom-search-input">
    <form method="get" action="/tienda">
      <div
        class="input-group col-xs-8 col-sm-6 col-md-4 col-lg-5 pull-right">
        <input type="text" class="search-query form-control"
          placeholder="Nombre de la canción" name="busqueda"/>
        <span class="input-group-btn">
          <button class="btn" type="submit">
            <span class="glyphicon glyphicon-search"></span>
          </button>
        </span>
      </div>
    </form>
  </div>
</div>

<h2>Canciones</h2>
<div class="row">

  <!-- http://librosweb.es/libro/bootstrap_3/capitulo_2/tipos_de_rejillas.html -->
  <!-- Inicio del Bloque canción -->
  {% for cancion in canciones %}
  <div class="col-xs-6 col-sm-6 col-md-4 col-lg-3">
    <div style="width: 200px">
      <a href="/cancion/id"> 
      <!-- http://www.socicon.com/generator.php -->
      <div class="wrap">{{ cancion.nombre }}</div>
      <div class="small">{{ cancion.autor }}</div>
      <div class="text-right">{{ cancion.precio }} €</div>
    </a>
    </div>
  </div>
  {% endfor %}
  <!-- Fin del Bloque canción -->
</div>

<!-- Paginación mostrar la actual y 2 anteriores y dos siguientes -->
<div class="row text-center">
  <ul class="pagination">
    <li class="page-item"><a class="page-link" href="#">Anterior</a></li>
    <li class="page-item active"><a class="page-link" href="#">1</a></li>
    <li class="page-item"><a class="page-link" href="#">2</a></li>
    <li class="page-item"><a class="page-link" href="#">3</a></li>
    <li class="page-item"><a class="page-link" href="#">4</a></li>
    <li class="page-item"><a class="page-link" href="#">5</a></li>
    <li class="page-item"><a class="page-link" href="#">Siguiente</a></li>
  </ul>
</div>
{% endblock %}
```



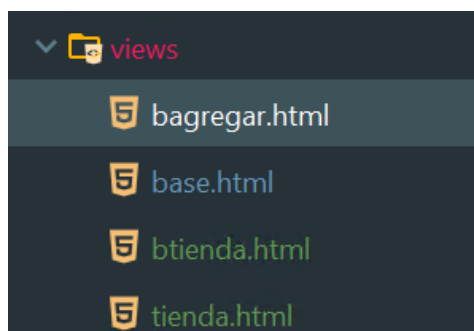
14 Vista para agregar canciones

Añadimos a **rcanciones.js** la posibilidad de responder a la petición **GET /canciones/agregar** que devolverá una vista **bagregar.html** con un formulario para añadir nuevas canciones.

```
module.exports = function(app, swig) {  
  app.get('/canciones/agregar', function (req, res) {  
    let respuesta = swig.renderFile('views/bagregar.html', {  
    });  
    res.send(respuesta);  
  })  
}
```

Nota: el método **GET /canciones/agregar** tiene que ir antes de **/canciones/:id**. El orden en el que escribimos los métodos GET y POST es importante puesto que **indican PRIORIDAD**. Si lo hacemos al revés, siempre que intentemos agregar una canción, es como si estuviéramos accediendo a la canción cuya id es "agregar".

Vamos a crear la vista **bagregar.html** en la carpeta **/views**. Obviamente, la nueva vista heredará de **base.html** y redefiniremos los bloques necesarios.





Nos fijamos en la URL contra la que se envía el formulario (**atributo action**) y las claves de los parámetros (**atributos name**).

```
{% extends "base.html" %}

{% block titulo %} Agregar canción {% endblock %}

{% block contenido_principal %}
<h2>Agregar canción</h2>
<form class="form-horizontal" method="post" action="/cancion">
  <div class="form-group">
    <label class="control-label col-sm-2" for="nombre">Nombre:</label>
    <div class="col-sm-10">
      <input type="text" class="form-control" name="nombre"
        placeholder="Nombre de mi canción" required="true" />
    </div>
  </div>
  <div class="form-group">
    <label class="control-label col-sm-2" for="genero">Genero:</label>
    <div class="col-sm-10">
      <select class="form-control" name="genero" required="true">
        <option value="pop">Pop</option>
        <option value="folk">Folk</option>
        <option value="rock">Rock</option>
        <option value="reagge">Reagge</option>
        <option value="rap">Hip-hop Rap</option>
        <option value="latino">Latino</option>
        <option value="blues">Blues</option>
        <option value="otros">Otros</option>
      </select>
    </div>
  </div>
  <div class="form-group">
    <label class="control-label col-sm-2" for="precio">Precio (€):</label>
    <div class="col-sm-10">
      <input type="number" class="form-control" name="precio"
        placeholder="2.50" required="true" />
    </div>
  </div>
  <div class="form-group">
    <label class="control-label col-sm-2" for="portada">Imagen portada:</label>
    <div class="col-sm-10">
      <input type="file" class="custom-file-input" name="portada" />
    </div>
  </div>
  <div class="form-group">
    <label class="control-label col-sm-2" for="audio">Fichero audio:</label>
    <div class="col-sm-10">
      <input type="file" class="custom-file-input" name="audio" />
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
      <button type="submit" class="btn btn-primary">Agregar</button>
    </div>
  </div>
</form>
{% endblock %}
```

Sí ejecutamos la aplicación y entramos en <http://localhost:8081/canciones/agregar> podremos ver el formulario.



Agregar canción

Nombre:

Genero:

Precio (€):

Imagen portada: Ningún archivo seleccionado

Fichero audio: Ningún archivo seleccionado

Nota: Subir el código a GitHub en este punto. Commit Message → *"SDI-NodeJS- 11,12,13,14 Vistas y motores de plantillas"*.

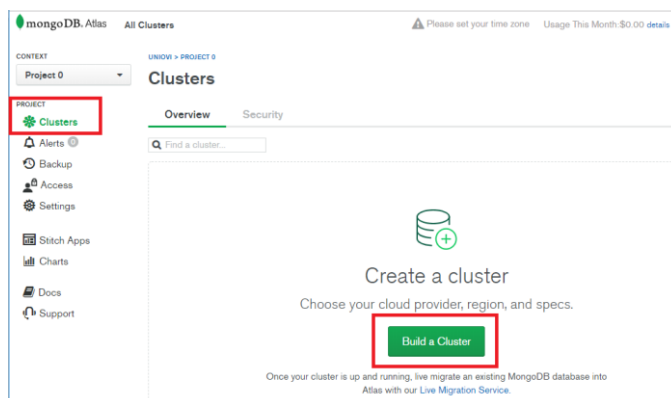
15 Mongo DB en la Nube (Obligatorio)

MongoDB es una base de datos no relacional, orientada a **documentos**, donde información se almacena siguiendo una estructura JSON. Internamente MongoDB maneja BSON, una versión más ligera de JSON en formato binario <https://www.mongodb.com/json-and-bson>. Ejemplo de documento en MongoDB:

```
{
  nombre: "Cambiar ordenadores",
  descripcion: "Cambiar todos los ordenadores del piso 1"
}
```

El proveedor <https://cloud.mongodb.com/> nos permite crear una instancia de MongoDB en la nube utilizando una cuenta gratuita (limitada a 500mb). Completamos el proceso de registro y nos identificamos en la plataforma.




Creemos un nuevo **Cluster**





Seleccionamos **AWS** como nuestro proveedor, en la región de Irlanda

Cloud Provider & Region AWS, Ireland (eu-west-1) ▾



Create a **free tier cluster** by selecting a region with **FREE TIER AVAILABLE** and choosing the **M0** cluster tier below.

★ recommended region ⓘ

NORTH AMERICA	EUROPE	AUSTRALIA
N. Virginia (us-east-1) ★ FREE TIER AVAILABLE	Stockholm (eu-north-1) ★	Sydney (ap-southeast-2) ★
Ohio (us-east-2) ★ FREE TIER AVAILABLE	Ireland (eu-west-1) ★ FREE TIER AVAILABLE	ASIA
N. California (us-west-1)	London (eu-west-2) ★	Tokyo (ap-northeast-1) ★
Oregon (us-west-2) ★ FREE TIER AVAILABLE	Paris (eu-west-3) ★	Seoul (ap-northeast-2)

Le damos el nombre **tiendamusica** y creamos el Cluster. Tardará unos 10 minutos en terminar de crear el cluster.

Cluster Name tiendamusica ▾

One time only: once your cluster is created, you won't be able to change its name.

Cluster names can only contain ASCII letters, numbers, and hyphens.

FREE Free forever! Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

Cancel Create Cluster

Accedemos a la opción **Database Access**, para añadir un nuevo usuario, pulsando en **Add New User**.

mongoDB Atlas All Clusters Usage This Month: \$0.00 details Preferences Miguel Sánchez-Santillán ▾

CONTEXT Project 0 ▾ MIGUEL'S ORG - 2019-03-18 > PROJECT 0

Database Access

ATLAS
Clusters
Data Lake BETA

SECURITY
Database Access
Network Access
Advanced

Database Users Database Roles

+ ADD NEW USER

User Name	Authentication Method	MongoDB Roles	Actions
-----------	-----------------------	---------------	---------



Creamos un nuevo usuario con privilegios **Atlas admin**, nombre **admin** y password **sdi** (recomendado otro password más complejo). Recordar el nombre de usuario y password ya que lo usaremos en la URI.

Add New User

SCRAM Authentication
SCRAM is MongoDB's default authentication method.

Username: admin
e.g. new-user_31
Password: sdi
SHOW

Autogenerate Secure Password

User Privileges

Atlas admin
Read and write to any database
Only read any database
Select Custom Role

Add Default Privileges

Save as temporary user

Cancel Add User

Normalmente deberíamos configurar las IPs que pueden conectarse a la base de datos. Por seguridad, solo debería tener acceso la IP del ordenador que ejecutara la aplicación. Accedemos a **Network Access** → **Ip WhiteList** y pulsamos el botón **"ADD IP ADDRESS"**.

mongoDB Atlas All Clusters Usage This Month: \$0.00 details Preferences Miguel Sánchez-Santillán

CONTEXT
Project 0

MIGUEL'S ORG - 2019-03-18 > PROJECT 0

Network Access

IP Whitelist Peering Private Endpoint

ADD IP ADDRESS

IP Address Comment

ATLAS
Clusters
Data Lake BETA

SECURITY
Database Access
Network Access
Advanced

Como en nuestro caso vamos a acceder desde diferentes sitios (casa, universidad) y no queremos perder tiempo configurando la IP, vamos a permitir conexiones desde todas. ¡Ojo! Esto es altamente inseguro para un desarrollo profesional.

Add Whitelist Entry

Add a whitelist entry using either CIDR notation or a single IP address. [Learn more.](#)

ADD CURRENT IP ADDRESS ALLOW ACCESS FROM ANYWHERE

Whitelist Entry: 0.0.0.0/0

Comment: Optional comment describing this entry

Save as temporary whitelist

Cancel Confirm



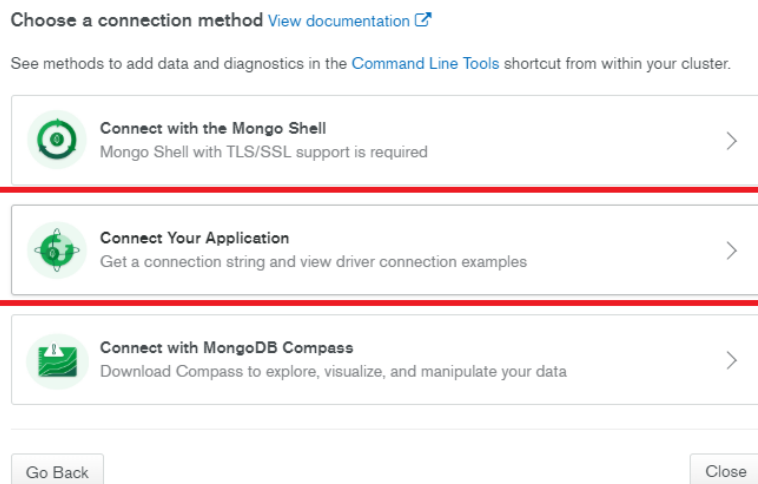
Para obtener la URL de conexión, volvemos a **Clusters** y hacemos click en **Connect**.



Seleccionamos el método de conexión pulsando en **Choose a connection method**.



Indicamos que nos vamos a conectar desde una Aplicación, **Connect Your Application**.





El driver que estamos usando es Node.js , la versión 2.

1 Choose your driver version

DRIVER: Node.js

VERSION: 2.2.12 or later

2 Add your connection string into your application code

Connection String Only Full Driver Example

```
mongodb://admin:<password>@tiendamusica-shard-00-00-wh9kn.mongodb.net:27017,tiendamusica-shard-00-01-wh9kn.mongodb.net:27017,tiendamusica-shard-00-02-wh9kn.mongodb.net:27017/test?ssl=true&replicaSet=tiendamusica-shard-00&authSource=admin&retryWrites=true
```

Replace **<password>** with the password for the *admin* user.
When entering your password, make sure that any special characters are [URL encoded](#).

Having trouble connecting? [View our troubleshooting documentation](#)

Go Back Close

Copiamos y guardamos la URL de conexión y la completamos con los datos de nuestro usuario.

```
mongodb://admin:sdi@tiendamusica-shard-00-00-wh9kn.mongodb.net:27017,tiendamusica-shard-00-01-wh9kn.mongodb.net:27017,tiendamusica-shard-00-02-wh9kn.mongodb.net:27017/test?ssl=true&replicaSet=tiendamusica-shard-00&authSource=admin&retryWrites=true
```

16 Instalación y verificación MongoDB Local (NO HACER)

En este apartado se instalará un servidor de base de datos MongoDB en local, que luego podrá ser utilizada para persistir datos en aplicaciones.

Para instalar Mongo descargamos el **Community Server** para Windows de la web oficial <https://www.mongodb.com/download-center> y seguimos los pasos para la instalación completa.



Creemos una estructura de carpetas **/data/db** en un directorio en el que tengamos permisos. Por ejemplo: **C:\Users\virtual_user\data\db** pero podría ser cualquier otro.

Desde la línea de comandos CMD accedemos a la ruta donde hayamos instalado MongoDB, por defecto: "C:\Program Files\MongoDB\Server\3.4\bin". Para arrancar el servidor ejecutamos el comando **mongod --dbpath C:\Users\virtual_user\data\db**.

¡Importante! El directorio debe coincidir con el de nuestra carpeta DB.

Cuando el servidor arranca de forma correcta debería mostrar el siguiente mensaje: **waiting connections on port 27017**.

```
2016-10-16T18:00:04.543+0200 I - [initandlisten] Detected data files in C:\data\db\ crea
storage engine, so setting the active storage engine to 'wiredTiger'.
2016-10-16T18:00:04.544+0200 I STORAGE [initandlisten] wiredtiger_open config: create,cache_si
viction=(threads_max=4),config_base=false,statistics=(fast),log=(enabled=true,archive=true,path
y),file_manager=(close_idle_time=100000),checkpoint=(wait=60,log_size=2GB),statistics_log=(wait
2016-10-16T18:00:04.716+0200 I NETWORK [HostnameCanonicalizationWorker] Starting hostname cano
2016-10-16T18:00:04.716+0200 I FTDC [initandlisten] Initializing full-time diagnostic data
:/data/db/diagnostic.data'
2016-10-16T18:00:04.718+0200 I NETWORK [initandlisten] waiting for connections on port 27017
```

Sí aparece algún problema al iniciar la base de datos debemos probar a utilizar el siguiente comando:

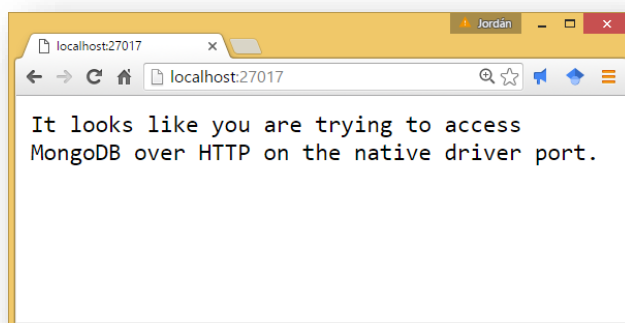
```
mongod --storageEngine=mmapv1 --dbpath C:\Users\virtual_user\data\db
```

Puede ser buena idea crear un script **iniciarBD.bat** para iniciar la base de datos, para las rutas anteriormente citadas el contenido sería el siguiente:

```
cd C:\Program Files\MongoDB\Server\3.4\bin
mongod --dbpath C:\Users\virtual_user\data\db
```

Para detener el servidor de bases de datos mongo debemos pulsar **Control + C**.

Para comprobar que el servidor listo y esperando por conexiones accedemos a <http://localhost:27017> desde el propio navegador.




















17 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.

Commits on Mar 16, 2019

SDI-NodeJS- 11,12,13,14 Vistas y motores de plantillas	 3de4204	
 edwardnunez committed 8 minutes ago		
SDI-NodeJS- 8,9,10 Recursos estáticos, Peticiones POST y parámetros, ...	 2bc69f0	
 edwardnunez committed an hour ago		

Commits on Mar 15, 2019

SDI-NodeJS- 6,7 División de las rutas en módulos, peticiones Get y pa...	 2e2b549	
 edwardnunez committed 2 hours ago		
SDI-NodeJS-.3,4,5 Express, depuración y variables de entorno	 030e9a4	
 edwardnunez committed 2 hours ago		
SDI-NodeJS 3.0 Creación del proyecto Tienda Virtual.	 c432281	
 edwardnunez committed 4 hours ago		

Commits on Mar 14, 2019

Initial commit	Verified	 9089efa	
 edwardnunez committed a day ago			