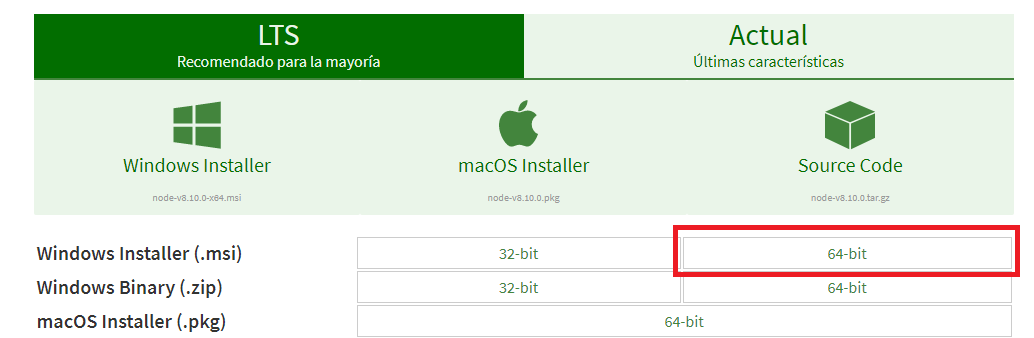
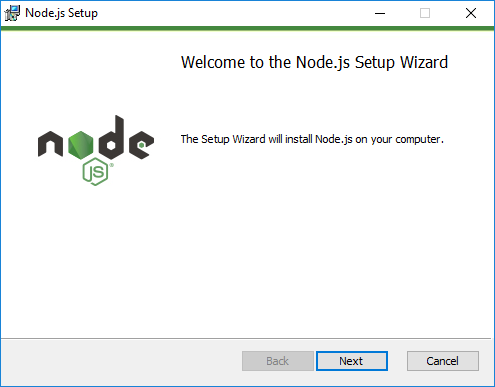
**Node JS - 1**

# Introducción

# Instalación de Node JS

Para trabajar con NodeJs debemos instalarlo. Descargamos el instalador de **Node 10 (Versión.\*)** en la siguiente URL <https://nodejs.org/es/download/>





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

# WebStorm

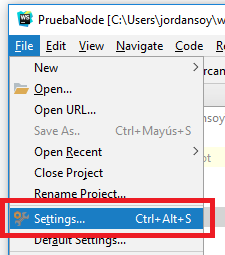
Descargamos el Webstorm <https://www.jetbrains.com/webstorm/>

Con la licencia de estudiante se puede usar de forma gratuita (de lo contrario es una versión de prueba valida durante 30 días)

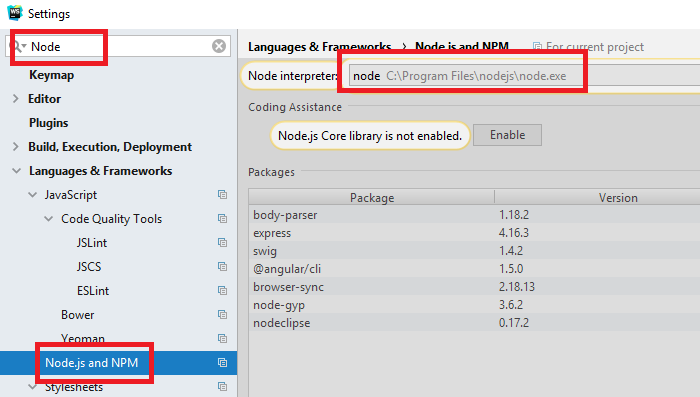
<https://www.jetbrains.com/student/>

## Validar la configuración de Node

Abrimos el Webstorm , Pulsamos en **File -> Open**



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



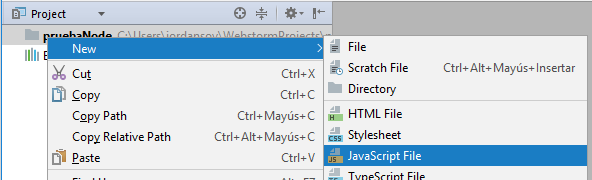
## Crear proyectos Node e importar el framework Express

Si queremos crear un proyecto Node vacío en WebStorm pulsamos en **File -> New Project**

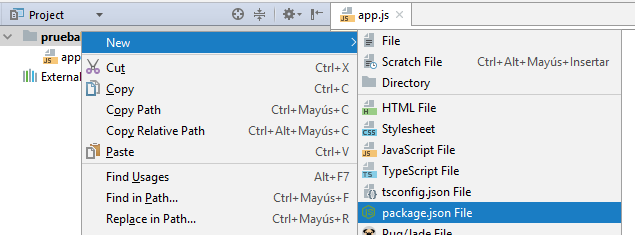
Seleccionamos el tipo **Empty Project**



Incluimos un nuevo fichero de tipo **JavaScript File** y lo llamamos **app.js**

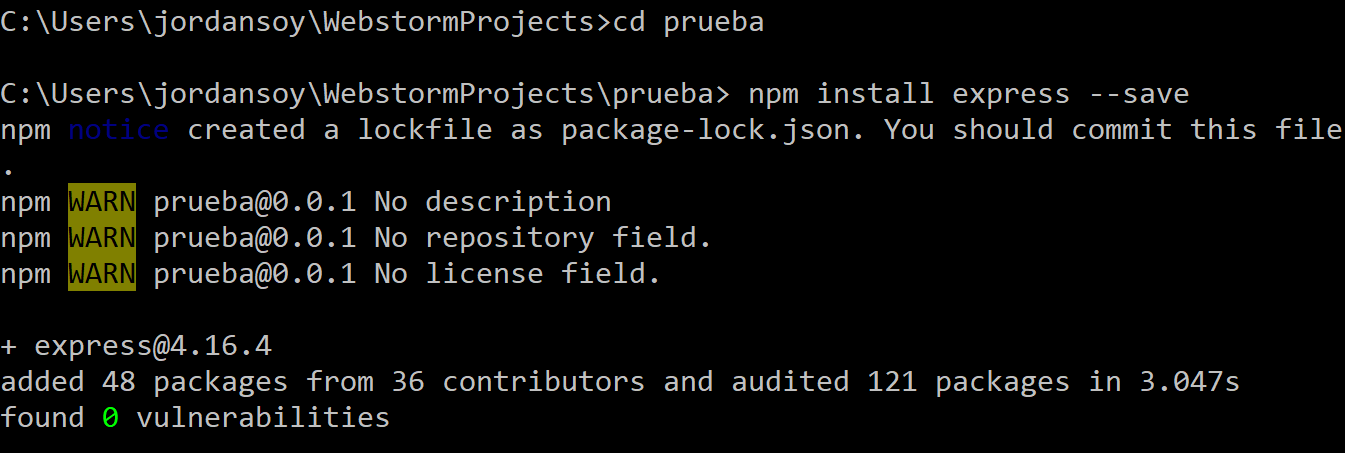


Incluimos un nuevo fichero de tipo **package.json**

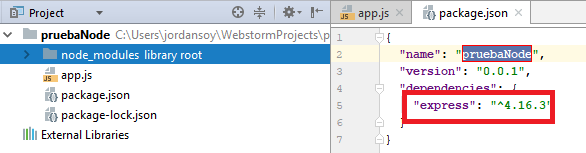


Para instalar un módulo abrimos el CMD, nos colocamos en la carpeta principal del proyecto y ejecutamos el comando del npm

**npm install express@4.16.1 --save**



Los ficheros relativos al módulo aparecerán en la carpeta **node\_modules** y la dependencia se habrá agregado al **package.json**



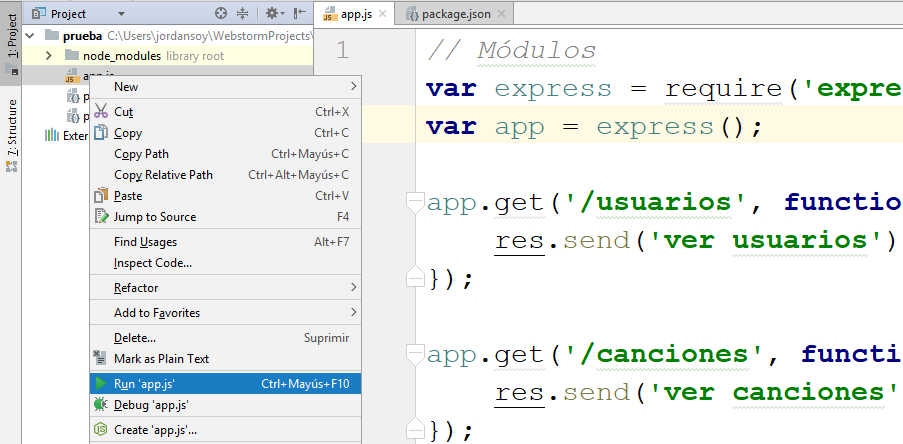
Incluimos el siguiente código en el fichero **app.js.** La función **require** se utiliza para introducir el uso de módulos, en este caso el módulo “express”, está función retornará la variable con el módulo express, para crear una nueva aplicación express ejecutamos la función **express()** que retorna la propia aplicación. La variable **app** será por lo tanto nuestra aplicación.

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.

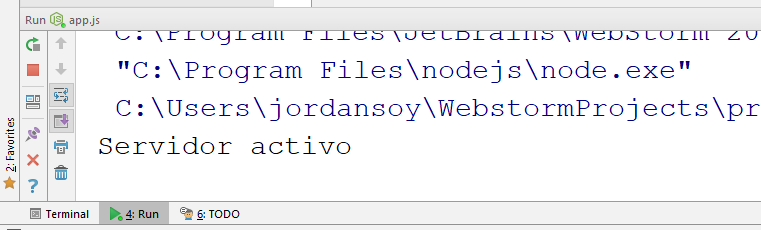
|  |
| --- |
| // Módulos  **var** express = require('express');  **var** app = express();  app.get('/usuarios', **function**(req, res) {  res.send('ver usuarios');  });  app.get('/canciones', **function**(req, res) {  res.send('ver canciones');  });  // lanzar el servidor  app.listen(8081, **function**() {  console.log("Servidor activo");  }); |

# Ejecución de la aplicación

Para ejecutar la aplicación botón derecho sobre el fichero principal **app.js** y pulsar en **Run** ‘**app.js**’



Nos indicará que la aplicación se desplego satisfactoriamente



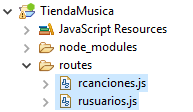
# Variables de entorno

Podemos guardar variables de entorno en la aplicación con **app.get** y **app.set** utilizando un String como clave, es una buena idea almacenar variables de configuración como el puerto.

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

# División de las rutas en módulos

Creamos la carpeta **/routes** y dentro de ella los ficheros **rusuarios.js**  y **rcanciones.js.** Cada uno de ellos será un **módulo como función**, que se encargará de gestionar las rutas de una entidad (usuarios o canciones). Para declarar un módulo utilizamos **module.exports** , el módulo puede recibir parámetros en su constructor, 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");  });  }; |

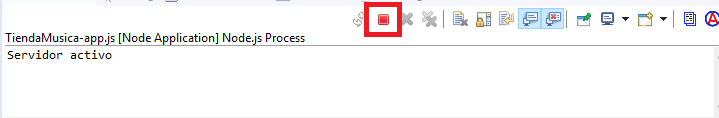
Finalmente eliminamos las respuestas get de **app.js** , las sustituiremos por el require de los módulos **rusuarios** y **rcanciones**, debemos enviar la variable **app** como parámetro en el require.

|  |
| --- |
| // 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");  }) |

En este caso estamos utilizando los módulos **rcanciones.js** y **rusuarios.js** como **funciones**, al realizar la llamada a la función se ejecuta todo su contenido.

Recordar siempre: Antes de cada ejecución debemos recordar guardar los cambios en todos los ficheros y detener la ejecución anterior.





# Peticiones GET y parámetros

Las peticiones Get pueden contener parámetros en su URL. Existen dos formas comunes de enviar parámetros (req – request), una de ellas es envía la clave y el valor del parámetro, 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](http://localhost:8081/canciones?nombre=despacito&autor=Luis%20Fonsi)

*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 Get enviados de este modo hace falta utilizar **req.query.<clave\_parámetro>** . Por ejemplo, para obtener el parámetro **nombre** y **autor** de la petición **/GET Cancion** deberíamos hacer lo siguiente:

|  |
| --- |
| module.exports = **function**(app) {  app.get("/canciones", **function**(req, res) {  **var** 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](http://localhost:8081/canciones?nombre=despacito&autor=Luis%20Fonsi)

Los parámetros son opcionales, en el caso de no encontrar los parámetros solicitados con **req.query.<clave\_del\_parámetro>.**  en la petición nos retornará “**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) {  **var** 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, si quisiéramos tratarlas como enteros habría que convertirlas. En el siguiente ejemplo se intenta sumar dos números que son enviados como parámetros.

|  |
| --- |
| module.exports = **function**(app) {  app.get("/canciones", **function**(req, res) {  **var** 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);  });  app.get('/suma', **function**(req, res) {  **var** respuesta = req.query.num1 + req.query.num2;  res.send(respuesta);  })  }; |

Sí guardamos los cambios y accedemos a <http://localhost:8081/suma?num1=19&num2=30> vemos que realmente no se están sumando, se están concatenando ya que son cadenas, habría que pasarlos a enteros ( **parseInt()** ), después deberíamos parsear a String la respuesta ya que **res.send** solo admite cadenas.

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

Otra forma de enviar parámetros GET es embeberlos en la URL entre /<valor\_parámetro>/ **sin especificar que Clave tienen**, se suele utilizar para ids y categorización, es la implementación del controlador la que determina la clave del parámetro en función de su posición en la URL.

* <http://localhost:8081/canciones/121/>

*Canción con parámetro “id” = 121*

* <http://localhost:8081/canciones/pop/121/>

*Canción con parámetro “genero” = pop e “id” = 121*

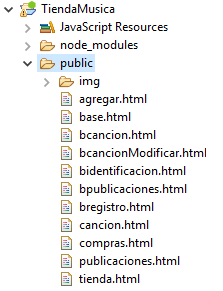
Estos parámetros se corresponderían con estas claves sí la aplicación así lo especificase, los parámetros se deben especificar el la ruta con **:<clave\_del\_parámetro>** y se accede a ellos en el código utilizando **req.params.<clave\_del\_parámetro>.**

|  |
| --- |
| app.get('/canciones/:id', **function**(req, res) {  **var** respuesta = 'id: ' + req.params.id;  res.send(respuesta);  })  app.get('/canciones/:genero/:id', **function**(req, res) {  **var** respuesta = 'id: ' + req.params.id + '<br>'  + 'Genero: ' + req.params.genero;  res.send(respuesta);  }) |

# Recursos estáticos

Express provee una función de asistencia (middleware) para facilitar el acceso a los clientes a ficheros estáticos, como pueden ser páginas HTML estáticas, imágenes, videos, css, etc. Basta con crear un directorio (por convenio se le suele llamar “**public”**) y declararlo en la aplicación, utilizando la función **express.static('<nombre del directorio>').**

Creamos la carpeta **public (por convenio se suele utilizar ese nombre para la carpeta publica)** en la raíz del proyectoy descomprimimos dentro el contenido del fichero **recursos.zip** descargado del campus virtual

****

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](http://localhost:8081/agregar.htmlhttp://localhost:8081/img/user.png)  
El contenido de las carpetas estáticas se puede modificar sin falta de reiniciar la aplicación.

# Peticiones POST y parámetros

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

El fichero localizado en la ruta <http://localhost:8081/agregar.html> define un formulario que envía una petición **POST /cancion** .

El formulario **agregar.html** contiene varios inputs con atributo **name** (toma los valores: nombre, genero, precio, portada y audio).

Ejemplo, fragmento de **agregar.html** donde se define la clave name para el campo precio.

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

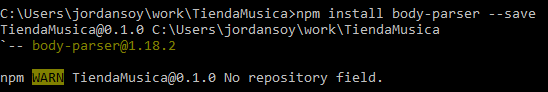
\*<https://www.w3schools.com/tags/tag_form.asp>

Vamos a hacer que nuestra aplicación procese la petición **POST /cancion**. Para poder acceder a los parámetros que se envían en el 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.



Ejecutamos el comando de instalación del módulo body-parser: **npm install body-parser@1.18.2 --save**



Volvemos al fichero principal **app.js** , primero nos aseguramos de añadir el require del módulo “**body-parse**”.

Después del require debemos declarar el uso del módulo con **app.use** del módulo (Referenciamos a las 2 funciones paseadoras **bodyParser.json()** (JSON) y **bodyParser.urlencoded()** (formularios estándar) ).

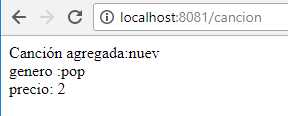
|  |
| --- |
| // Módulos  **var** express = require('express');  **var** app = express();  **var** bodyParser = require('body-parser');  app.use(bodyParser.json());  app.use(bodyParser.urlencoded({ extended: **true** }));  app.use(express.**static**('public')); |

A partir de este momento podemos acceder a al cuerpo del body utilizando **req.body.<nombre\_del\_parámetro>.** Abrimos el fichero **rcanciones.js** Creamos una función **app.post** que procese los atributos nombre, género y precio.

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

Guardamos los cambios, ejecutamos la aplicación, utilizamos el formulario <http://localhost:8081/agregar.html> para comprobar que la petición POST está funcionando correctamente **no agregar los ficheros**.





# 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, etc.

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

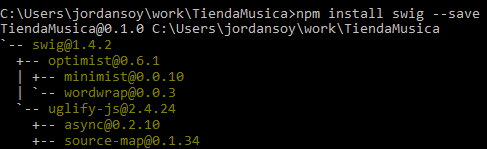
Lista de ejemplos de enrutamiento con comodines y expresiones regulares: <http://expressjs.com/es/guide/routing.html> .

# Vistas y Motores de plantillas

Una de las formas más comunes para intercalar datos obtenidos por medio de la ejecución de lógica de negocio en ficheros de presentación HTML son las plantillas. Las plantillas nos permiten combinar texto de salida (por ejemplo HTML) con tratamiento de datos procedentes de la lógica , aplicando difernetes etiquetas, funciones y filtros.

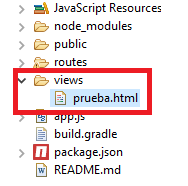
Podemos combinar el framework express con muchos motores de plantillas. **Swig** es un motor con un buen nivel de funcionalidad y con bastantes similitudes a **twig** (muy popular en otras tecnologías) <http://node-swig.github.io/swig-templates/>

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



Creamos una nueva carpeta **/views** en el directorio raíz, dentro creamos un fichero muy simple **prueba.html** (las plantillas pueden tener cualquier extensión incluyendo .html , no tienen una extensión especifica como en otros entornos).

No es nada recomendable almacenar vistas en el directorio **/public**



La plantilla va a contener código HTML y también atributos que van a ser enviados desde el controlador a la vista. Estos atributos serán:

* **vendedor**: parámetro de tipo cadena con el nombre del vendedor.
* **canciones**: lista de objetos de tipo canción, 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 % }**

Implementamos la plantilla **prueba.html**

|  |
| --- |
| <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...* [http://paularmstrong.github.io/swig/docs/tags/](http://paularmstrong.github.io/swig/docs/tags/%20). <http://node-swig.github.io/swig-templates/docs/tags/>

Ahora vamos a utilizar la plantilla en la aplicación. Tenemos que importar el módulo utilizando **require('swig')**. Esta función nos devuelve el objeto **swig** capaz de renderizar plantillas.

|  |
| --- |
| // Módulos  **var** express = require('express');  **var** app = express();  **var** swig = require('swig');  **var** bodyParser = require('body-parser');  app.use(bodyParser.json());  app.use(bodyParser.urlencoded({ extended: **true** }));  app.use(express.**static**('public')); |

Vamos a enviar la variable **swig** como parámetro a los módulos que implementan los controladores (para que puedan usar el motor de plantillas).

|  |
| --- |
| //Rutas/controladores por lógica  require("./routes/rusuarios.js")(app, swig);  require("./routes/rcanciones.js")(app, swig); |

Debemos incluir el parámetro en ambos módulos **rcanciones** y **rusuarios**.

|  |
| --- |
| 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");  });  }; |

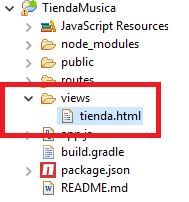
Cuando se reciba una petición **GET /canciones** , crearemos un array con 3 canciones, cada una con nombre y precio. Finalmente ejecutamos la función **renderFile** del objeto **swig**. La función renderFile recibe como parámetros: la plantilla y los parámetros que se le envían ( string **vendedor** y array de **canciones**), el retorno de la función será el HTML generado por la plantilla, el cual podemos enviar ya como respuesta (**res.send**).

|  |
| --- |
| 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/prueba.html', {  vendedor : 'Tienda de canciones',  canciones : canciones  });  res.send(respuesta);  }); |

Sí guardamos los cambios, ejecutamos la aplicación y accedemos a [http://localhost:8081/canciones](http://localhost:8081/canciones%20)  deberíamos ver la plantilla correctamente renderizada.



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



Este fichero contiene el catálogo de una tienda y utiliza el framework **bootstrap** <https://getbootstrap.com/docs>. Por el momento solo contiene código HTML, analizamos su contenido e introducimos las etiquetas de script **swig** para mostrar la lista de canciones.



Analizamos el fichero **tienda.html** y localizamos el bloque donde se introduce la información del anunció.



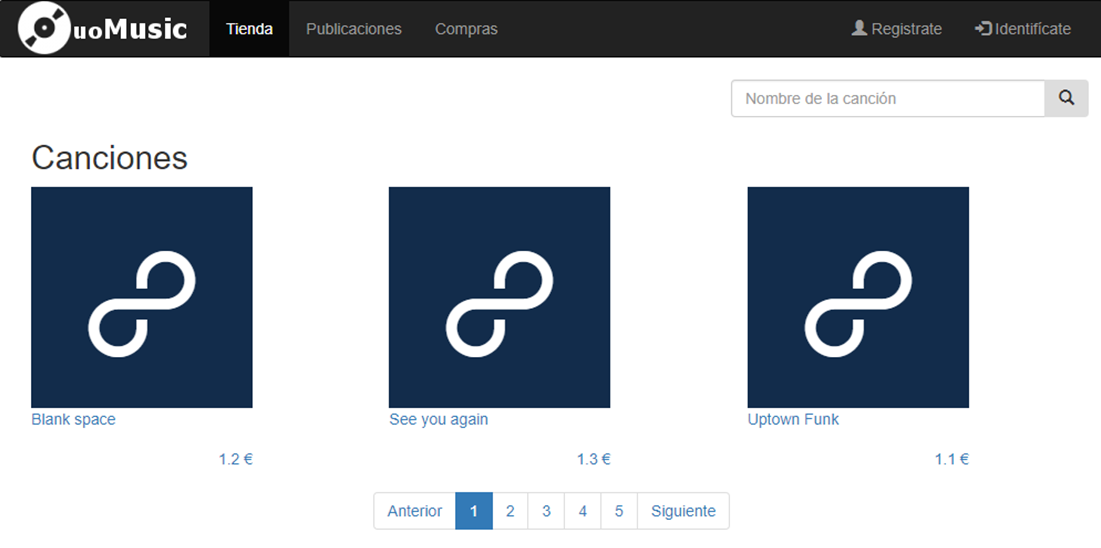
Introducimos el script que recorre la lista de **canciones** y muestra su información por el momento no tenemos toda la información, nos faltan el **autor** pero lo incluimos igualmente.

|  |
| --- |
| <!-- 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"*>  <img src=*"img/cancion1.png"* alt=*"logo"*/>  <!-- 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 --> |

Accedemos la función **Get /canciones** de **rcanciones.js** y modificamos la vista que se utilizará como base de la respuesta.

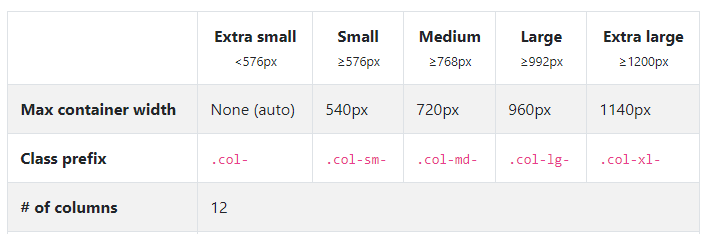
|  |
| --- |
| **var** respuesta = swig.renderFile('views/tienda.html', {  vendedor : 'Tienda de canciones',  canciones : canciones  }); |

Guardamos los cambios y comprobamos que el resultado es correcto.



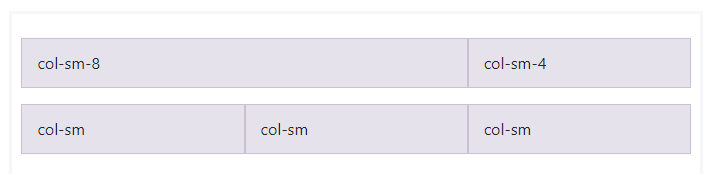
Sistema de Grid de Bootstrap <https://getbootstrap.com/docs/4.0/layout/grid/>

Permite especificar 4-5 distribuciones de columnas para diferentes tipos de pantalla (no hace falta especificarlas todas, si no se especifica extiende la inmediatamente inferior)



12 celdas para dividir, podemos repartir 12 unidades entre un número N de columnas. (Por ejemplo 2 columnas con 8 y 4)

Sí no ponemos nada se dividirán de forma proporcional (por ejemplo 3 columnas iguales)



Sí algo ocupa más de 12 va a saltar a la siguiente fila.

**Mix and match**

Podemos especificar de 1 a 5 divisiones de columnas distintas dependiendo de la pantalla.



* En resoluciones 1) Extra Small 2) Small la primera columna ocupa 12
* En resoluciones 3) Medium , 4) Large y 5)Extra Large la primera ocupa ocupa 8

|  |  |
| --- | --- |
| idea icon | **Elementos clicables reconocibles**  En todos los sitios web se debe identificar claramente los elementos en los que el usuario puede hacer click.  Convendría remarcar el aspecto visual de las cajas para poder identificar rápidamente que elementos son clicables. |

# Plantillas – URLs absolutas

Es muy común que las plantillas accedan a recursos estáticos (css , imágenes, fuentes, ficheros js como jquery, ficheros html ) , estos recursos se almacenan en un **directorio estático** (hemos creado previamente el directorio **public**).

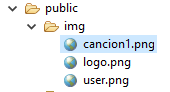
En nuestro caso **tienda.html** accede a dos recursos estáticos, el icono de la barra de navegación, verificar:

<img src=*"img/logo.png"* alt=*"logo"*/>

Tambien las caratulas de las canciones.

<img src=*"img/cancion1.png"* alt=*"logo"*/>

Al cargar la respuesta de **localhost:8081/canciones** , el navegador busca en ***localhost:8081/img/logo.png*** y ***localhost:8081/img/cancion1.png*** como hemos copiado todos los recursos gráficos de tipo imagen en la carpeta **public/img**/ el recurso se localiza con éxito.



Sin embargo, que pasaría si la URL base fuera **localhost:8081/nuevas/canciones**

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

Especificar las imágenes **img/logo.png** e **img/canción1.png** de forma relativa la aplicación buscaría los recursos en **localhost:8081/nuevas/img/logo.png ,** en ese directorio no existe el recurso.Una buena práctica para prevenir problemas de este tipo suele ser usar rutas absolutas. Modificamos el documento para introducir rutas absolutas.

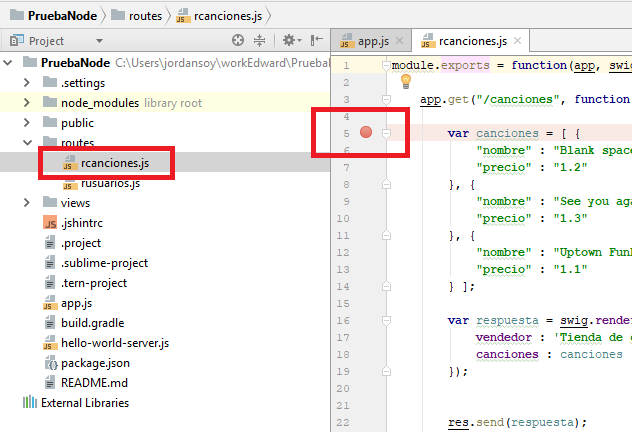
|  |
| --- |
| <img src=*"/img/logo.png"* alt=*"logo"*/> |

|  |
| --- |
| <img src=*"/img/cancion1.png"* alt=*"logo"*/> |

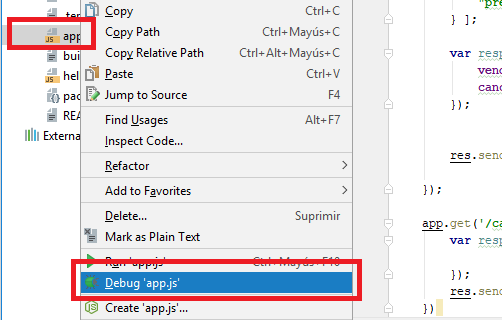
**\*Nota:** los cambios en las plantillas no se reconocen de forma dinámica, hay que volver a desplegar la aplicación.

Depurar

Una vez cargado seleccionamos cualquier fichero .js y añadimos los puntos de ruptura.



Para ejecutar en modo depuración pulsamos botón derecho sobre el fichero principal, **app.js** y la acción **Debug ‘app.js’**



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

Al hacer la petición que haga entrar a la petición en un punto de ruptura, se detendrá la ejecución, por ejemplo <http://localhost:8081/canciones>

Desde la pestaña **Debugger** podemos 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

# 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, para ello no copiamos el mismo código en todas las vistas (sería poco eficiente y muy costoso realizar cambios) utilizamos una plantilla base con bloques identificados y redefinición de bloques.

Copiamos en la carpeta **/view** el fichero **base.html** que se encuentra actualmente en **/public/**.Es básicamente una página sin contenido.



Dentro de **base.html** vamos a identificar los bloques donde se deberían agregar contenidos, utilizando las etiquetas **{% block <id\_del\_bloque %} {% endblock %}**, podemos identificar tantos bloques como queramos dentro de la página, nos interesara incluir bloques en todas las zonas que vayan a poder ser modificadas en otras partes de las 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** de contenido.

|  |
| --- |
| <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 **título**.

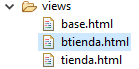
|  |
| --- |
| <!DOCTYPE html>  <html lang=*"en"*>  <head>  <title>{% block titulo %} uoMusic {% endblock %}</title>  <meta charset=*"utf-8"*/>  <meta name=*"viewport"* content=*"width=device-width, initial-scale=1"*/> |

Suele ser de mucha utilidad incluir un bloque en la cabecera para incluir más css o ficheros js, puede ser que puntualmente alguna página necesite incluir nuevos scripts o css, llamamos 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**. Los cuales pueden ser redefinidos (o no) en todas las plantillas que heredan de **base.html**

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



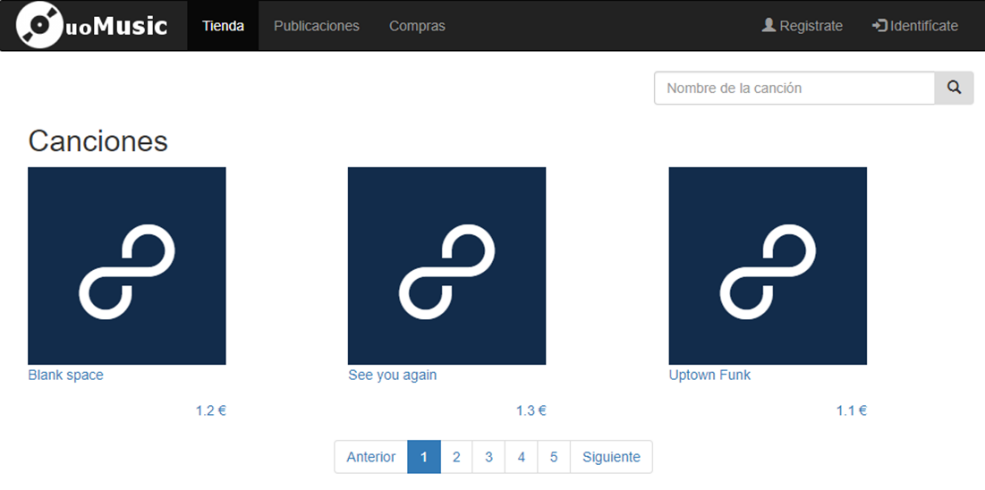
Abrimos el fichero **btienda.html** y agregamos la directiva **{% extends “base.html” %}**, a partir de aquí incluimos solo la redefinición de los bloques que nos interesen **{% block <nombre\_del\_bloque> %}** **contenido redefinido** **{% endblock %}** definimos el contenido para los bloques **título** y **contenido principal**. En 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"*> <img src=*"img/cancion1.png"* alt=*"logo"* />  <!-- 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.



**\*Nota:** en la versión anterior y para simplificar habíamos utilizado una versión “incompleta” de la vista, le agregamos la barra de búsqueda y la paginación, aunque aún no tendrán funcionalidad.

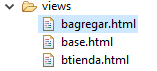
|  |
| --- |
| {% block contenido\_principal %}  <!-- Búsqueda -->  <div class=*"row"*>  <div id=*"custom-search-input "*>  <form method=*"get"* action=*"/tienda"* >  <div  class=*"input-group col-xs-8 col-sm-6 col-md-4 col-lg-5 pull-right"*>  <input type=*"text"* class=*"search-query form-control"*  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"*> <img src=*"img/cancion1.png"* alt=*"logo"* />  <!-- 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 %} |

# Vista para agregar canciones

La aplicación responderá a la petición **GET /canciones/agregar** enviando una vista **bagregar.html** donde se mostrará un formulario que solicitará los datos de la canción.

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

Para crea la vista **bagregar.html** nos vamos a basar en la plantilla **base.html**. Al usar una plantilla base solo tenemos que redefinir el bloque del contenido principal. El contenido de **bagregar.htlm** será el siguiente:

****

Nos fijamos en la URL contra la que se envía el formulario y las claves de los parámetros.

|  |
| --- |
| {% 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.



# Mongo DB en la Nube

MongoDB es una base de datos no relacional, orientada a **documentos**, la información en los documentos almacena siguiendo una estructura JSON, internamente MongoDB maneja BSON, se trata de una versión más ligera en formato binario creada a partir del formato JSON [https://www.mongodb.com/json-and-bson](https://www.mongodb.com/json-and-bson%20%20).

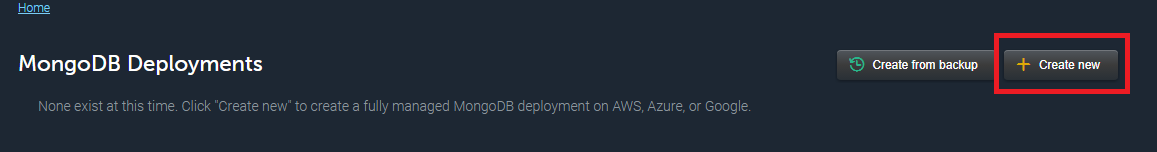
Ejemplo de documento proyecto en MongoDB. Ejemplo de documento en MongoDB:

|  |
| --- |
| **{**  nombre**:** "Cambiar ordenadores"**,**  descripcion**:** "Cambiar todos los ordenadores del piso 1"  **}** |

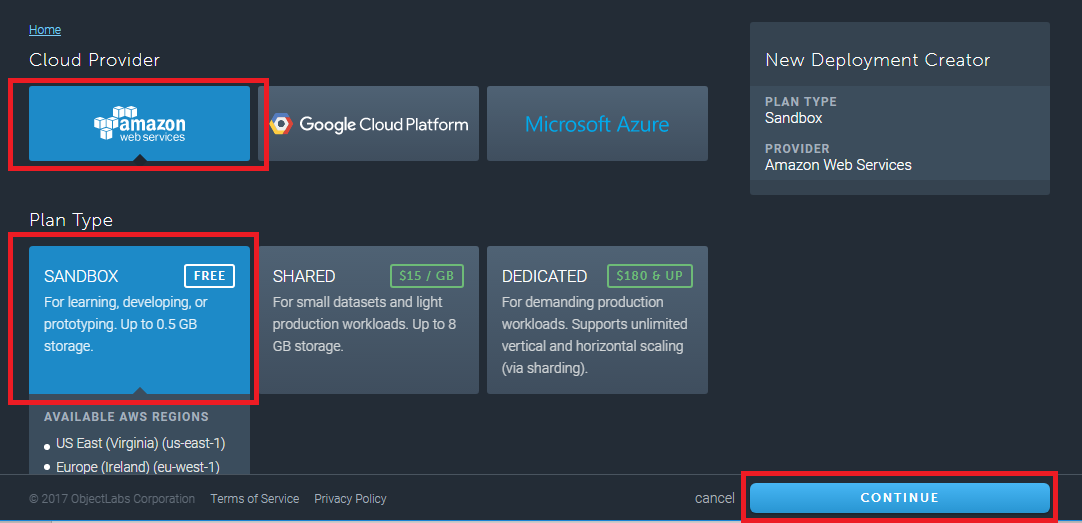
El proveedor <https://mlab.com/> nos permite crear una instancia de mongoDB en la nube utilizando una cuenta gratuita, esta versión de la cuenta está limitada a 500mb.

Completamos el proceso de registro y nos identificamos en la plataforma.

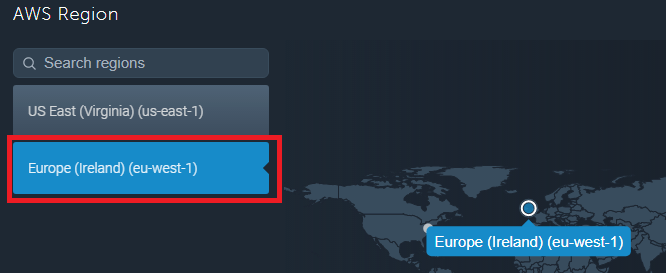
Creamos una nueva base de datos en la sección **MongoDB Deployments.**



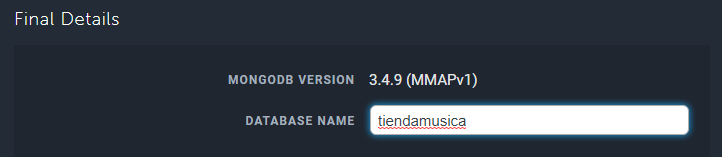
Seleccionamos **amazon** como nuestro proveedor



Seleccionamos Europa como **AWS Region**.

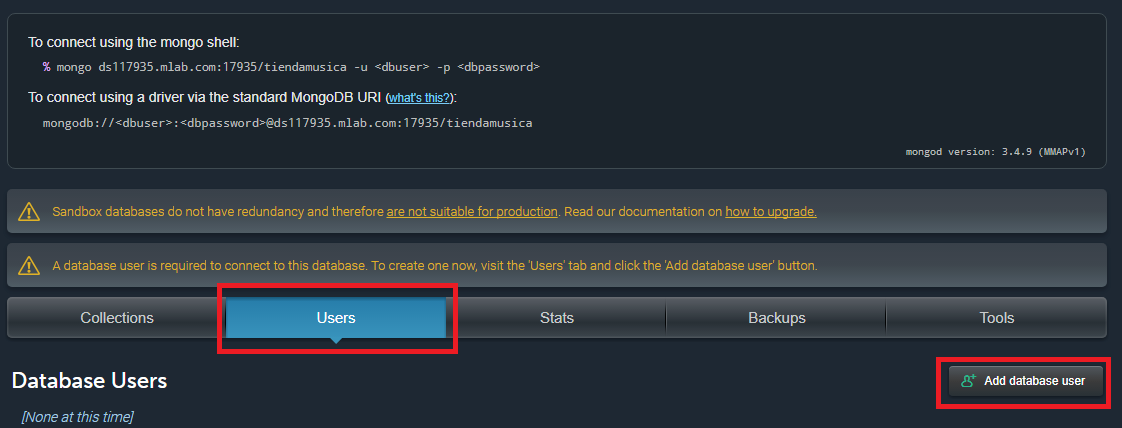


Le damos el nombre **tiendamusica** a la base de datos.



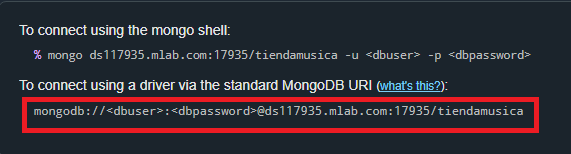
Una vez enviada la orden de creación la base de datos estará activa en unos segundos.

Seleccionamos la base de datos y creamos un nuevo usuario, nombre **admin** y password **admin** (recomendado otro password más complejo).



Usamos este nuevo usuario para completar la **URI** de conexión.



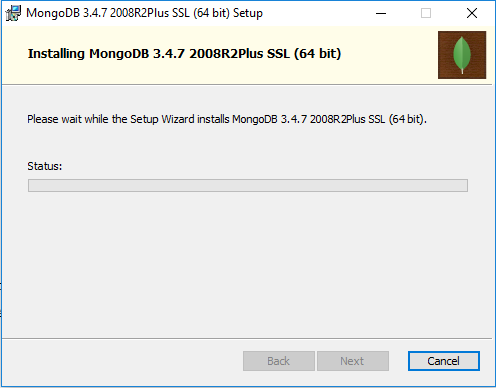


En este ejemplo la URI de conexión será: *mongodb://admin:sdi@ds117935.mlab.com:17935/tiendamusica*

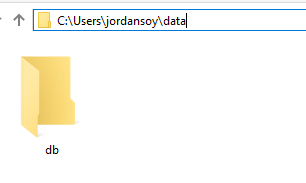
# Instalación y verificación de mongo DB local **(alternativa no hacer)**

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.



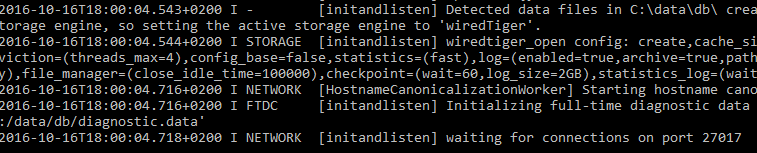
Creamos una estructura de carpetas **/data/db** en un directorio en el que tengamos permisos.



En este caso: **C:\Users\jordansoy\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\jordansoy\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**.



|  |  |
| --- | --- |
| Sí aparece algún problema al iniciar la base de datos debemos probar a utilizar el siguiente comando:  mongod --storageEngine=mmapv1 --dbpath C:\Users\jordansoy\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\jordansoy\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](http://localhost:27017%20) desde el propio navegador.

