

# Sistemas Distribuidos e Internet

## Tema 7

### Introducción a Node JS



# Índice

- Introducción a Node JS
- Principales ventajas y características de Node JS
- Arquitectura Node JS
- NPM (Node Package Manager)
- Entorno de desarrollo
- Aplicación
  - Estructura
  - Server
  - Dependencias
  - Despliegue
- Express
  - Instalación
  - Aplicación
  - Routing (Enrutamiento)
  - Peticiones Web
  - Recursos estáticos
  - Vistas y Plantillas

# ¿Que es Node js?

- Es una plataforma de software que permite ***ejecutar JavaScript del lado del servidor***
- Utilizando una Arquitectura:
  - ***Basada en eventos - (Event Loop)***
  - Gestión de operaciones de ***I/O se forma asíncrona y sin bloqueo.***
    - Utiliza un ***único hilo de ejecución (single threading)*** que gestiona las entradas y salidas asíncronas de los clientes conectados.
- Mediante su diseño modular, permite construir aplicaciones web rápidas y escalables



# ¿Que es Node js?

- Node nace en 2009 de la mano del desarrollador Ryan Dahl
- Actualmente es desarrollado por la Node.js Foundation
- Patrocinado por la empresa Joyent Inc, especializada en virtualización y computación en la nube

# ¿Que NO es Node JS?

- No es un servidor web
  - Contiene una biblioteca de servidor HTTP integrada
  - Permite el desarrollo de aplicaciones web con servidor integrado
  - Por lo que no necesita ejecutar un servidor web independiente como Apache o IIS.
- No es un lenguaje
  - Las aplicaciones se desarrollan usando JavaScript
- No es un Framework
  - Permite desarrollar e integrar frameworks, ejemplo express
- Node JS no es para aplicaciones multi-hilos.
- ***Node JS es una plataforma de software***

# Principales ventajas de Node JS

- Utiliza un único lenguaje de programación para desarrollar aplicaciones completas (frontend y backend)
  - Ejemplo: **JAVASCRIPT**
- Utiliza de **motor JavaScript V8** desarrollado por Google para el navegador Chrome y es extremadamente rápido.
- **Mejora la concurrencia** de acceso a servidor mediante:
  - Su Arquitectura es **Single-Thread with Event Loop**
    - Usa un modelo de operaciones **I/O asíncrono sin bloqueo**
    - Utiliza **un único hilo de ejecución** que gestiona las entradas y salidas asíncronas.

# Principales ventajas de Node JS

- Permite desarrollar sitios web donde prima la *eficiencia y la escalabilidad*.
- Formas de escalar cualquier aplicación:
  - **Vertical**
    - Consiste en agregar más recursos a un solo nodo.
  - **Horizontal**
    - Consiste en agregar más nodos a un sistema.
- El Node JS usa la *escalabilidad horizontal* en lugar de la escalabilidad vertical para las aplicaciones.

# Principales ventajas de Node JS

- Node JS es muy ligero y fácil de extender su funcionalidad
  - Por su diseño modular
- Es una buena opción para aplicaciones que han de procesar grandes volúmenes de datos en tiempo real.
- Buena integración con bases de datos no relacionales
  - Ejemplo: Mongo, Apache Casandra, etc.



# Principales ventajas de Node JS

- Node JS soporta muchos *motores de plantillas* :
  - JADE, swig, ejs, pug, Thymeleaf, etc.
- Tiene una API incorporada para desarrollar o crear *servidores HTTP, servidores DNS, servidores TCP*, etc.
- Ideal para *desarrolladores FullStack*.
- Es multiplataforma y de código abierto.
  - Mac OS, Windows, Linux, etc.
- Comunidad muy activa.
- Otros proyectos similares:
  - Tornado (Python), Jetty (Java), Twisted (Python), EventMachine (Ruby), etc.

# Principales características de Node JS

## ■ Basado en un diseño modular

- Cada funcionalidad es dividida en **módulos** o paquetes separados.
- Estos módulos permiten extender sus funcionalidades básicas.
- Cuando se instala, Node JS se incluye por defecto un conjunto de módulos (core)
  - Se puede añadir módulos adicionales
- Los módulos se pueden agregar de forma sencilla.
- La gestión de dependencia o paquetes en Node se realiza con ***NPM (Node Package Manager)***.

# Principales características de Node JS

## ■ NPM (Node Package Manager)

- Es el *gestor de paquetes* para JavaScript y Node.
- Facilita a los desarrolladores de JavaScript *reutilizar el código* que otros desarrolladores han compartido.
- Hay más de 600,000 paquetes de código JavaScript disponibles para descargar.
- NPM está escrito en Node JS, por lo que su sistema necesita tener instalado Node JS
- En la instalación de Node JS, por defecto se instala NPM.
- Página oficial:
  - <https://docs.npmjs.com/>

# Principales características de Node JS

- **Instalación de módulos usando npm (II)**
  - Los módulos se descargan e instalan *localmente* mediante el comando

```
npm install <package_name>
```

- Esto creará el directorio *node\_modules* en el directorio actual (si no existe) y descargará el paquete a ese directorio.

# Principales características de Node JS

- **Declarar el uso de una paquete en una aplicación**
  - Es necesario que el paquete este instalado previamente.
  - Incluir el paquete el fichero *package.json* de la aplicación.
    - Por defecto, cuando se instala un paquete se añade a este fichero como dependencia de la aplicación.



```
package.json
1 {
2   "name": "TiendaMusica",
3   "version": "0.1.0",
4   "description": "TiendaMusica",
5   "main": "hello-world-server.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified!\"",
8   },
9   "repository": "",
10  "keywords": [
11    "node.js",
12    "eclipse",
13    "nodeclipse"
14  ],
15  "author": "",
16  "license": "MIT",
17  "readmeFilename": "README.md",
18  "dependencies": {
19    "body-parser": "^1.18.2",
20    "express": "^4.16.2",
21    "express-fileupload": "^0.3.0",
22    "express-session": "^1.15.6",
23    "mongodb": "^2.2.33",
24    "swig": "^1.4.2"
25  }
26 }
```

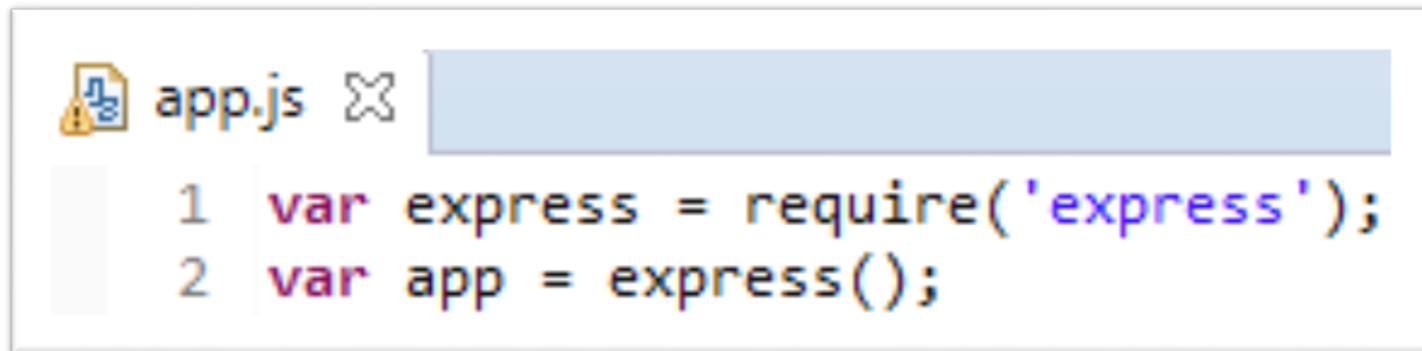
# Principales características de Node JS

- **Usar un paquete en una aplicación Node JS (II)**

- En el fichero js correspondiente, se añade el paquete usando la siguiente sintaxis:

```
var my_package = require('<package_name>')
```

- Ejemplo:



The screenshot shows a code editor window with a tab labeled 'app.js'. The code inside the editor is as follows:

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

# Principales características de Node JS

- *Algunos comandos importantes de npm*

Comando	Descripción
<b>npm install &lt;package_name&gt;</b> Ej. npm install express	Instala las dependencias en la carpeta local del proyecto <b>node_modules</b> <u>Por defecto instala la ultima versión de la dependencia</u>
<b>npm install &lt;package_name&gt; -g</b> Ej: npm install grunt -g	Instala las dependencias en el directorio de trabajo como un paquete global
<b>npm install &lt;package_name&gt; --no-save</b>	No agrega la declaración de la dependencia al <b>package.json</b>
<b>npm install &lt;package_name&gt; --save</b>	Agrega la dependencia al <b>package.json</b>
<b>npm install &lt;name&gt;@&lt;version&gt;</b> Ej. npm install express@4.16.2	Instala la versión especificada de la dependencia

# Principales características de Node JS

- **Modelo de operaciones I/O asíncrona o sin bloqueo (Non-blocking)**
  - Node JS usa el *modelo de I/O asíncronico* para realizar tareas complejas como:
    - Leer o escribir en el sistema de archivos.
    - Almacenar información en Bases de datos
    - Establecer comunicación de red o comunicarse con otros componentes
  - Estas operaciones se delegan directamente al SO o BD.
  - Esto permite ejecutar varios procesos de E/S de forma simultánea sin producir un bloqueo en el sistema.

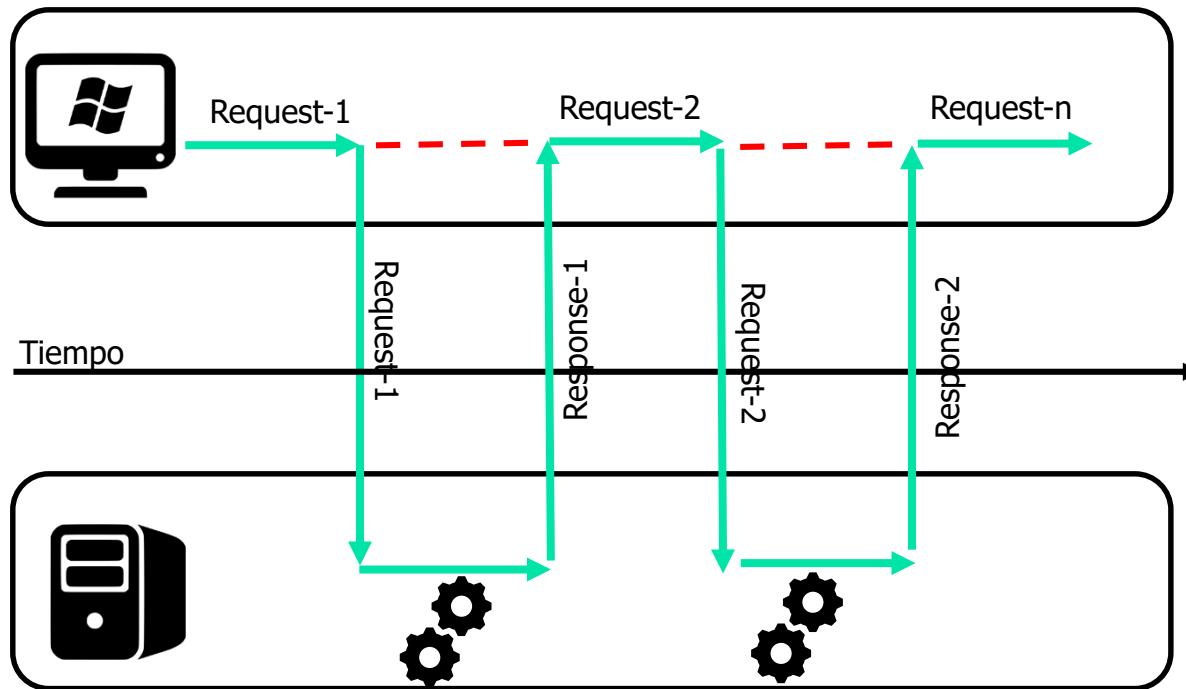


# Principales características de Node JS

- **Características de una comunicación *síncrona***
  - Las operaciones son bloqueantes(blocking).
  - Las operaciones se ejecutan de forma secuencial.
  - El programa permanece bloqueado hasta que termine la operación.
  - El cliente espera la respuesta del servidor para continuar con el flujo del programa.

# Principales características de Node JS

## ■ Funcionamiento comunicación síncrona



--- Tiempo de espera (Bloqueo)

***Comunicación síncrona***


# Principales características de Node JS

## ■ Ejemplo comunicación síncrona en Javascript

```
<script>
var ages = [8, 10, 15, 25, 65, 23, 18, 55, 75, 88, 77, 99, 100]
function Search(minAge, maxAge) {
  console.log('START search with age between:', minAge, "and", maxAge);
  result = ages.filter(function(age) {
    return age >= minAge && age <= maxAge;
  });
  return result;
}


function executeSearch() {
  var maxAge = 0;
  var maxRange = 4;
  for (var i = 0; i < maxRange; i++) {
    minAge = maxAge;
    maxAge = maxAge + 25;
    result = Search(minAge, maxAge);
    console.log('END search with age between:', minAge, "and",
      maxAge, ' and result =', result);
  };

  console.log('EXECUTION COMPLETED');
}
</script>
```



# Principales características de Node JS

## ■ Ejemplo comunicación síncrona en Javascript > Resultado



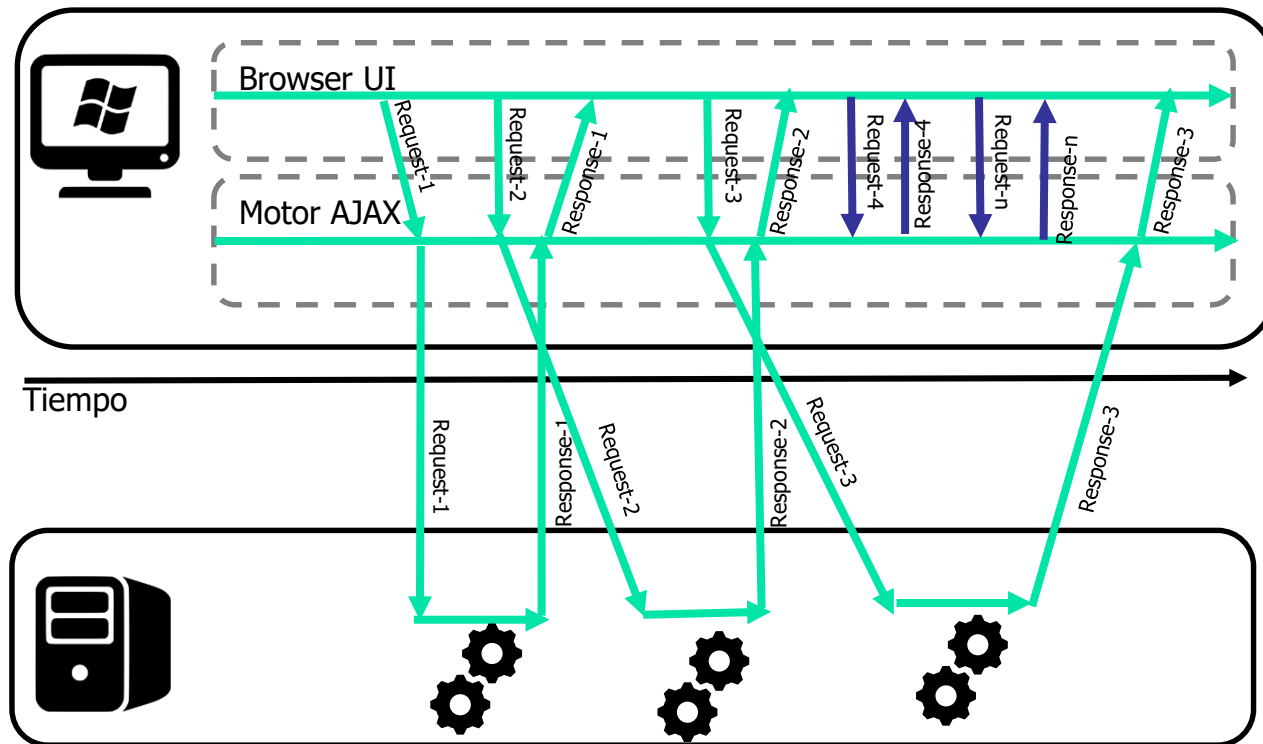
```
START search with age between: 0 and 25
END search with age between: 0 and 25 and result = ► (6) [8, 10, 15, 25, 23, 18]
START search with age between: 25 and 50
END search with age between: 25 and 50 and result = ► [25]
START search with age between: 50 and 75
END search with age between: 50 and 75 and result = ► (3) [65, 55, 75]
START search with age between: 75 and 100
END search with age between: 75 and 100 and result = ► (5) [75, 88, 77, 99, 100]
EXECUTION COMPLETED
```

# Principales características de Node JS

- **Características de una comunicación *asíncrona***
  - Las operaciones *no son bloqueantes*.
    - No se espera a que una operación termine para continuar con el flujo del programa
  - Normalmente se realizan mediante el sistema de *callback(retrollamadas) y/o promesas*.
  - En node JS, si un proceso de I/O tarda mucho tiempo, entonces permite que continúe otro proceso antes de que la transmisión haya finalizado.

# Principales características de Node JS

## ■ Funcionamiento comunicación asíncrona



***Comunicación asíncrona  
(Modelo AJAX)***

# Principales características de Node JS

- **Ejemplo Comunicación asíncrona en JavaScript usando *callback***

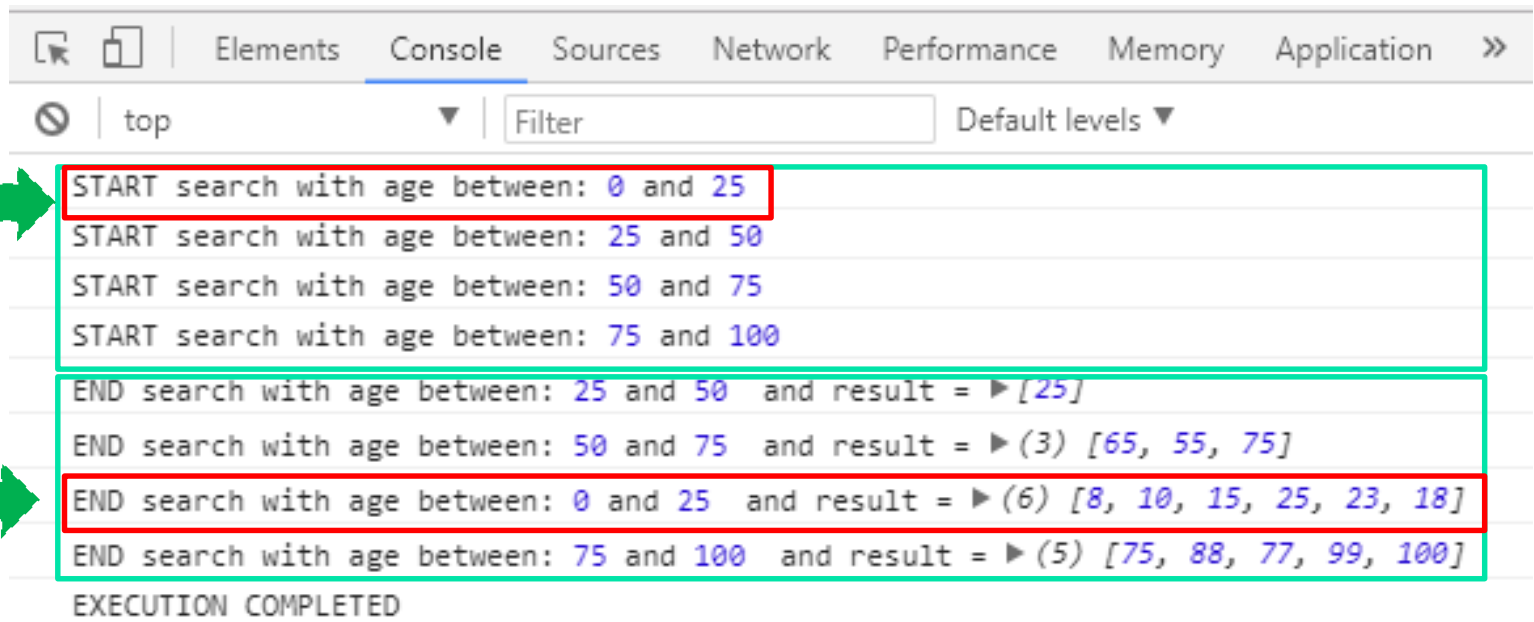
```
<script>
var ages = [8, 10, 15, 25, 65, 23, 18, 55, 75, 88, 77, 99, 100]
function asyncSearch(minAge, maxAge, callback) {
  console.log('START search with age between:', minAge, "and", maxAge);
  setTimeout(function() {
    result = ages.filter(function(age) {
      return age >= minAge && age <= maxAge;
    });
    callback(minAge, maxAge, result);
  }, 0 | Math.random() * 2000);
}

function executeAsyncSearch() {
  var maxAge = 0;
  var maxRange = 4;
  var count = 0;
  for (var i = 0; i < maxRange; i++) {
    minAge = maxAge;
    var maxAge = maxAge + 25;
    asyncSearch(minAge, maxAge, function (minAge, maxAge, result) {
      console.log('END search with age between:', minAge, "and", maxAge,
        ' and result =', result);

      if (++count === maxRange) {
        console.log('EXECUTION COMPLETED');
      }
    });
  }
}
</script>
```

# Principales características de Node JS

- **Ejemplo Comunicación asíncrona en JavaScript usando *callback* > Resultado**



```
START search with age between: 0 and 25
START search with age between: 25 and 50
START search with age between: 50 and 75
START search with age between: 75 and 100

END search with age between: 25 and 50 and result = ► [25]
END search with age between: 50 and 75 and result = ► (3) [65, 55, 75]
END search with age between: 0 and 25 and result = ► (6) [8, 10, 15, 25, 23, 18]
END search with age between: 75 and 100 and result = ► (5) [75, 88, 77, 99, 100]

EXECUTION COMPLETED
```

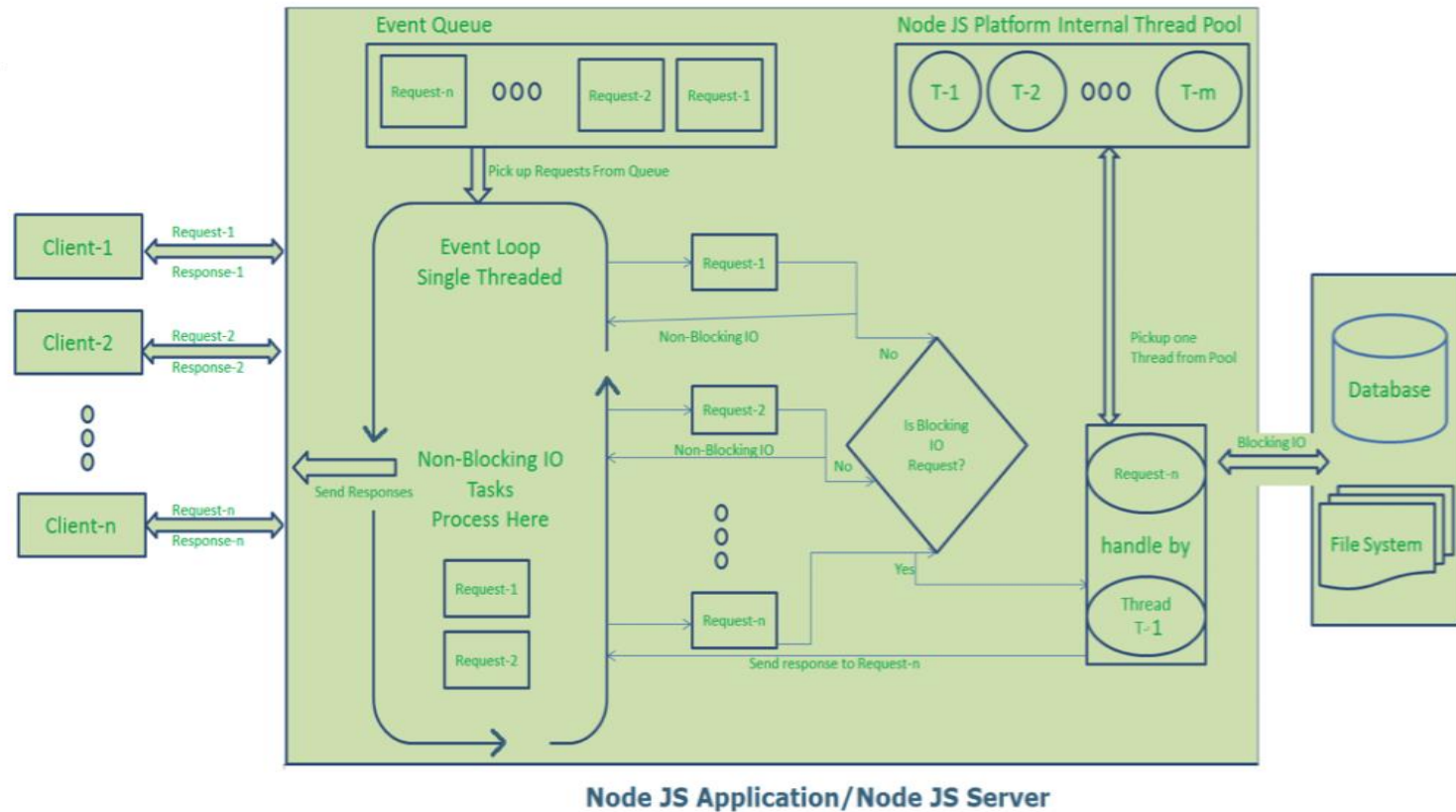


# Principales características de Node JS

## ■ Arquitectura basado en eventos

- Node se basa en la arquitectura **"Single Threaded Event Loop"**
- Utiliza **un único hilo de ejecución** que gestiona las peticiones concurrentes de los clientes.
- Es lo que permite el procesamiento **asíncrono de operaciones I/O**.
- Mejora la concurrencia de acceso a servidor mediante su **Bucle de Eventos (Event Loop)**.
- En cada petición realizada por un cliente, Node JS no genera un nuevo hilo, sino, que disparará un evento dentro del Event Loop.
- El modelo de procesamiento de Node JS se basa principalmente en el **modelo de eventos de JavaScript**, mediante el sistema de **callback(retrollamadas)**.

# Arquitectura Node JS



**Fuente:** <https://www.journaldev.com>

# ¿Cuándo usar o No Node JS?

## ■ Usar

- Cuando se necesitan mantener una conexión persistente entre el navegador y el servidor.
- Cuando se necesite realizar muchas operaciones de I/O de manera simultánea .
- Ideal para aplicaciones en tiempo real, como chats y juegos online, herramientas de colaboración, etc.
- Para el desarrollo de aplicaciones web con bases de datos NO relacionales.

## ■ No usar

- En aplicaciones que hagan usos intensivo CPU o de recursos de sistemas operativos.
  - Ejemplo, procesamiento de cálculos pesados.

# ¿Quiénes usan Node JS?

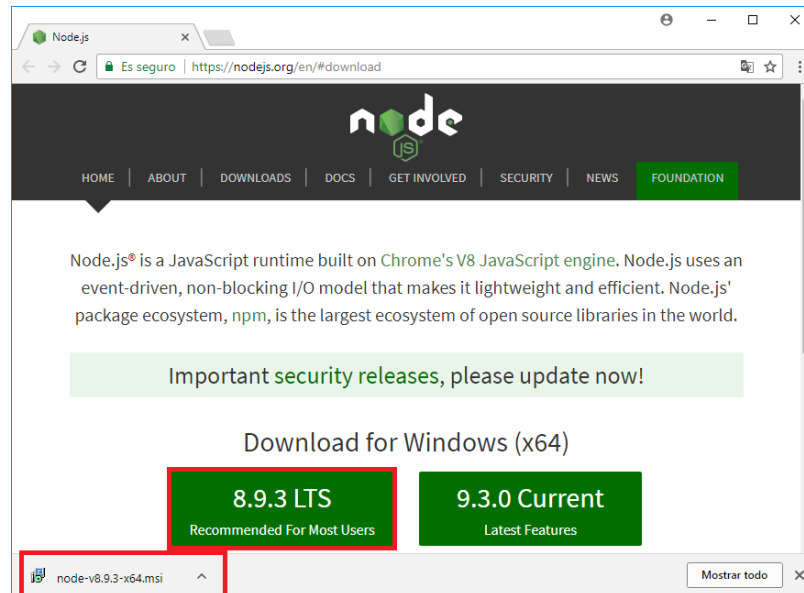


# Entorno de desarrollo

- Requisitos
  - Descargar e instalar Node.js en el sistema operativo.
- Pueden desarrollarse aplicaciones prácticamente en cualquier IDE
  - Notepad, Visual Studio, Eclipse, IntelliJ IDEA, Spring Tool Suite, etc.
  - Si utilizas un IDE, instalar y habilitar el plugin de Node JS.

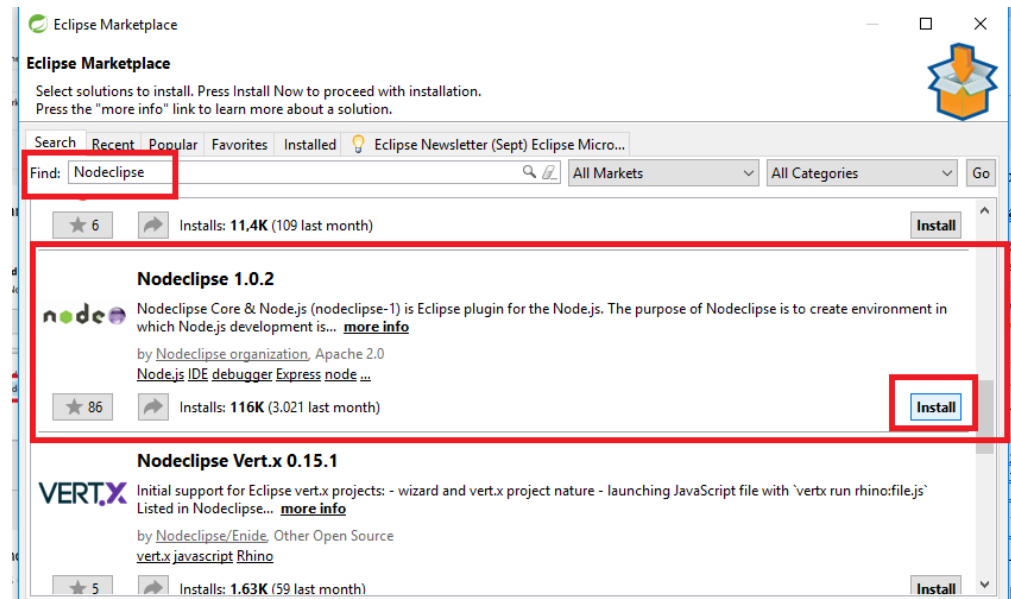
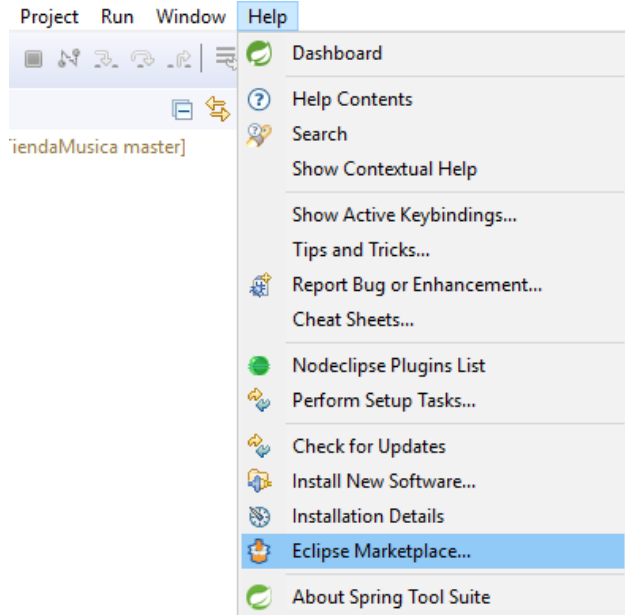
# Entorno de desarrollo > Eclipse

- Descargar e instalar Node.js en el sistema operativo
  - <https://nodejs.org/es/download/package-manager/>



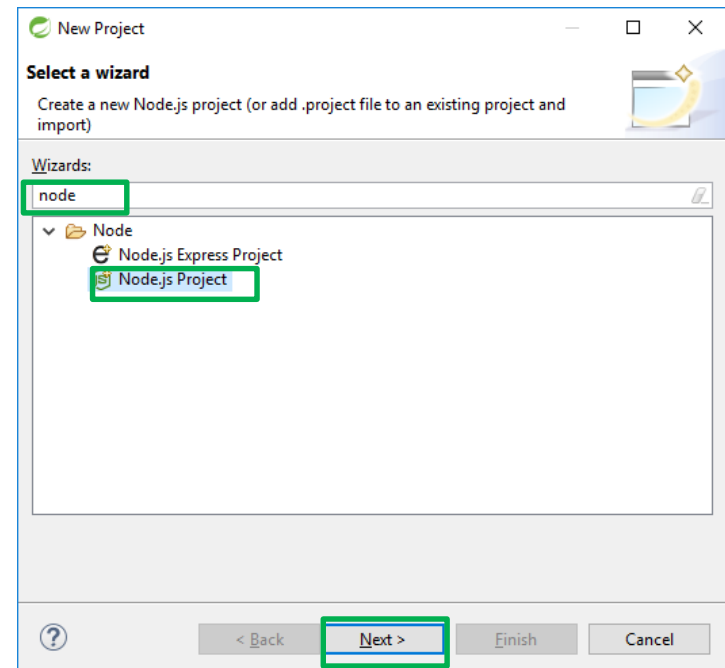
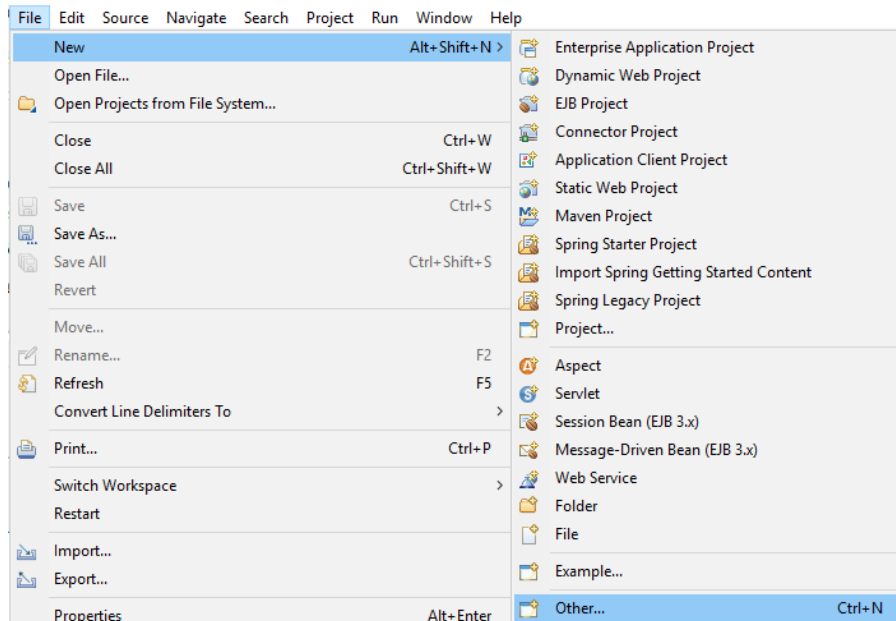
# Entorno de desarrollo > Eclipse

- Trabajar con Node JS en IDE de Eclipse o STS
  - Instalar y habilitar el **plugin Nodeclipse** del eclipse Marketplace.



# Aplicación

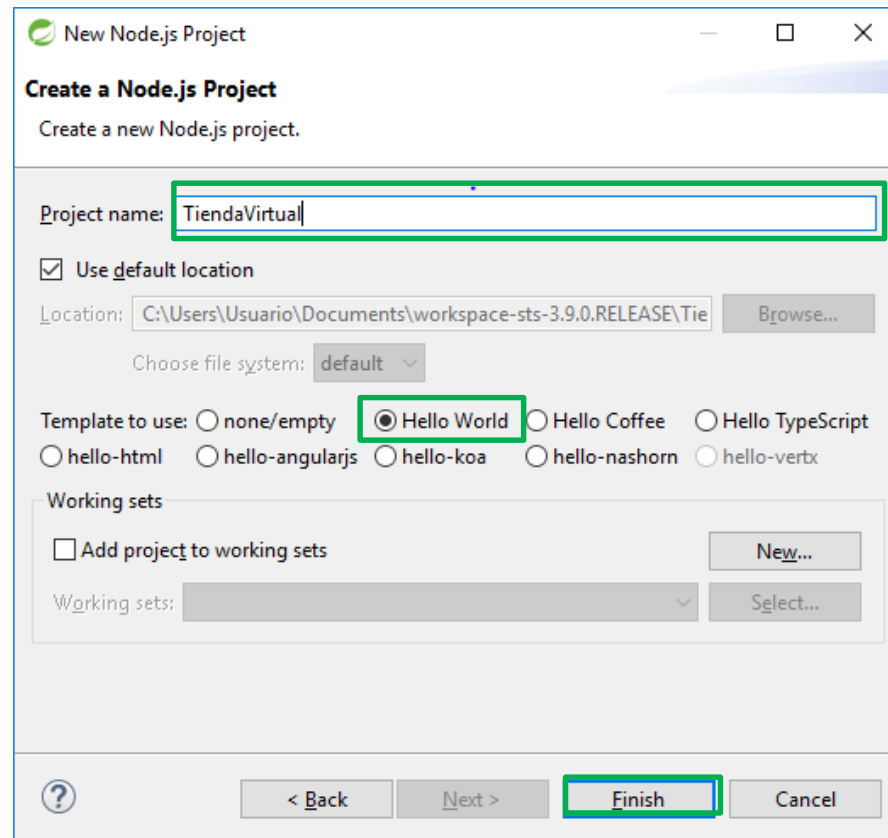
- Ejemplo creación de una aplicación web Node en eclipse





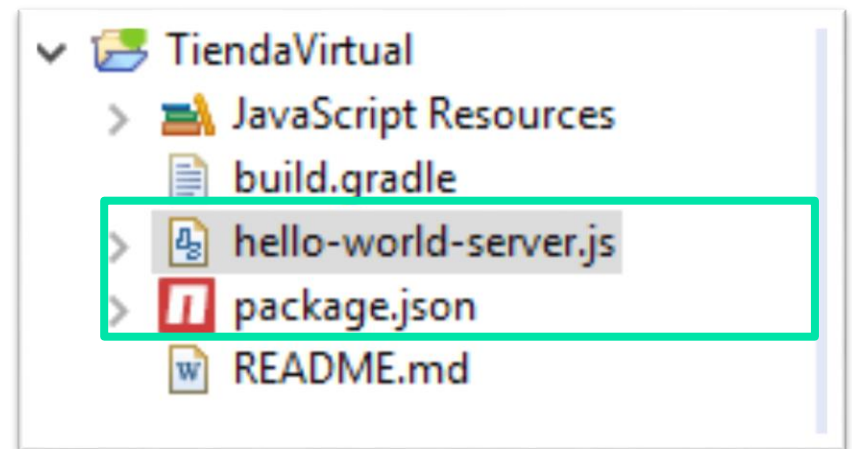
# Aplicación

- Ejemplo creación de una aplicación web Node en eclipse



# Aplicación > Estructura

- Hemos elegido una plantilla que crea una estructura básica
  - Otras plantilla incluyen código más complejo
- Los ficheros principales son:
  - El fichero ***hello-world-server.js*** contiene la ***lógica de la aplicación***.
  - fichero ***package.json*** define la ***configuración y los metadatos*** de la aplicación



# Aplicación > Server

- **El fichero hello-word-server.js**

- Contiene la *lógica de la aplicación*
- Inicialmente se define una variable que incluye el **módulo http** que viene incluido con Node.js
- Este módulo permite crear aplicaciones muy simples
  - En lugar de este módulo usaremos **express**

```
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/');
```

# Aplicación > Gestión de dependencia

- En cada aplicación Node JS debe haber un archivo **package.json** en la carpeta raíz de la aplicación.
- En este se definen:
  - La **configuración y los metadatos** de la aplicación
  - Las **dependencias** utilizadas
- No es obligatorio especificar las dependencias utilizadas
  - Pero si muy recomendable
- Las dependencias se instalan usando el **npm**
  - Por defecto se añaden a la lista de "dependencies"

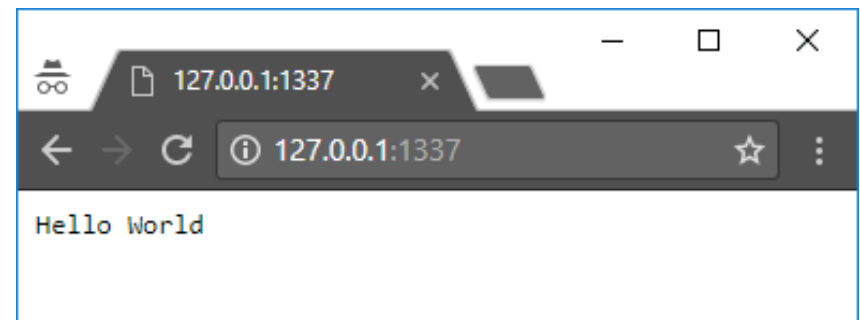
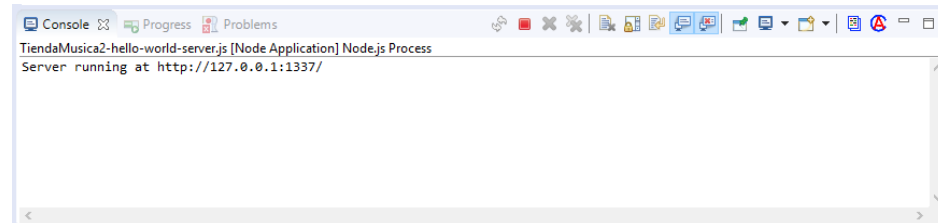
```
package.json
1 {
2   "name": "TiendaMusica",
3   "version": "0.1.0",
4   "description": "TiendaMusica",
5   "main": "hello-world-server.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\"",
8   },
9   "repository": "",
10  "keywords": [
11    "node.js",
12    "eclipse",
13    "nodeclipse"
14  ],
15  "author": "",
16  "license": "MIT",
17  "readmeFilename": "README.md",
18  "dependencies": {
19    "body-parser": "^1.18.2",
20    "express": "^4.16.2",
21    "express-fileupload": "^0.3.0",
22    "express-session": "^1.15.6",
23    "mongodb": "^2.2.33",
24    "swig": "^1.4.2"
25  }
26 }
```

**Metadatos**

**Dependencias**

# Aplicación > Despliegue

- Marcamos el fichero **hello-word-server.js** y lo ejecutamos, pulsamos el botón derecho y **Run as -> Node Application**
- 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



# Express

- Es un framework de desarrollo de aplicaciones web minimalista y flexible para Node JS.
- Proporciona mecanismos para:
  - Direccionamiento de URL (Routing) manejo de solicitudes HTTP con soporte a sus distintos metodos (Get, Post, Put, Delete, etc.)
  - Permite trabajar con distintos motores de plantillas (Jade, EJS, JinJS ...)
  - Establecer la configuración común de la aplicación web, como el puerto que se utilizará para la conexión, y la ubicación de las plantillas, etc.

# Express > Instalación

- El módulo express no pertenece al Core de Node JS
  - Módulos del Core: <https://nodejs.org/api/modules.html>
- Para instalar express use el comando

```
npm install express --save
```

- **--save** hace que express se declare en la lista de dependencias **package.json**
- Si Express no estaba instalado, se añade una nueva carpeta **node\_modules** con el código de los módulos

# Express > Estructura Aplicación

- **Estructura de directorios de una aplicación con express**
  - Esta es una “estándar”
    - Se puede cambiar según nuestras necesidades

```
Application
├─ app.js
├─ package.json
├─ public
│   └─ images
│   └─ javascripts
│   └─ stylesheets
│       └─ style.css
├─ routes
│   └─ index.js
│   └─ users.js
└─ views
    └─ error.html
    └─ index.html
```



# Express > Aplicación

Application

└─ app.js

...

## ■ App.js

- Entrada de la aplicación.
- La función *require* se utiliza para introducir *módulos*, en este caso express
- La función *express()* crea una **nueva aplicación express**
- La aplicación express se configura por medio de funciones. Ej:
  - **get**: especifica como responder a peticiones get en una URL
  - **listen**: inicia la escucha de peticiones en el puerto 8081.

// Módulos

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

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

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

// lanzar el servidor

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

# Express > Aplicación

## ■ Uso de variables de aplicación

- Permite declarar variables que pueden ser usadas en cualquier parte de la aplicación
  - EJ: referencia al puerto, directorios, etc.
- ***app.set***
  - Guarda una variable
  - Utiliza un string como clave
- ***app.get***
  - Recuperar el valor de una variable
  - Utiliza un string como clave

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

app.set('port', 8080);

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

# Express > Routing

## ■ Routing (enrutamiento)

- Definición de puntos finales de una aplicación (URI). Define como se procesa la petición (**request**) y la respuesta que se envía al cliente (**response**).
- Define:
  - Una **URI** o vía de acceso
  - Un **método** de solicitud HTTP
- Un enrutamiento se define:

***app.METHOD(PATH, HANDLER)***

Donde:

- **app**: es una instancia de una aplicación express.
- **METHOD**: es un método de solicitud HTTP
- **PATH**: es una vía de acceso al servidor.
- **HANDLER**: es la función que se ejecuta cuando se correlaciona la ruta.

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

app.get('/', function(req, res) {
  res.send('ver pagina de inicio');
});
```

# Express > Routing

## ■ Method (Método):

- Se corresponde con un método de solicitud HTTP
- Soporta múltiples métodos HTTP
- Ejemplo, como definir algunos métodos get y post

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

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

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

app.post('/cancion', function(req, res) {
    res.send(' registrar canción');
});

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

# Express > Routing

## ■ Path (rutas)

- En combinación con el método `http` definen los puntos finales (endpoints) a los que los clientes pueden hacer peticiones.
- Las vías de acceso pueden ser:
  - Cadenas(string)
  - Patrones de cadenas
  - Expresiones regulares
- Ejemplos ***cadenas***
  - El primer ejemplo responde a una solicitud GET en la pagina inicial de la aplicación usando la cadena `"/"`.
  - Los siguientes ejemplos responden en la ruta `"/cancion"` a las peticiones GET, POST

```
app.get('/', function(req, res) {  
    res.send('ver pagina de inicio');  
});  
  
app.get('/cancion', function(req, res) {  
    res.send('ver cancion');  
});  
  
app.get('/cancion/:id', function(req, res){  
    res.send('Info cancion');  
});  
  
app.post('/cancion', function(req, res) {  
    res.send(' registrar canción');  
});  
  
app.put('/cancion', function(req, res) {  
    res.send('actualizar canción');  
});
```

# Express > Routing

## ■ Enrutamiento y comodines

- Las rutas, se admite el uso de comodines ( ?, +, \* y () ) y otras expresiones regulares
- Ejemplo basado en Patrones de cadenas
  - Ej: cualquier ruta que **comience con promo:** /promo, /promocion, /promocionar

```
// Ejemplo basados en patrones string
// app.get('/promo*', function(req, res) {
//     res.send('Resp al patrón promo*');
// });
```

# Express > Routing

## ■ Manejadores (Handler)

- Funciones que se ejecuta cuando se recibe la petición
- Tienen acceso al objeto petición (req) y al objeto respuesta (res)
- Suelen :
  - Acceden a los parámetros de la petición
  - Invocar funcionalidad relativa a la lógica de negocio
  - Generar una respuesta.
  - Otros

```
app.get('/canciones', function (req, res){  
  // Obtener parámetro GET clave id  
  var id = req.query.id;  
  
  // Lógica de negocio  
  var cancion = gestor.getCancion(id);  
  
  // Composición y retorno de respuesta  
  var respuestaHTML =  
    swig.renderFile('views/cancion.html',  
      { cancion : cancion });  
  
  res.send(respuestaHTML);  
})
```

# Express > Routing

## ■ Métodos de respuesta (Objeto response)

- Son los métodos que **envían la respuesta** al cliente y terminan con el ciclo de petición/respuesta.
- Hay que invocarlo desde un manejador de rutas(handler).
- Si no se invoca la solicitud quedará pendiente.

```
app.get('/', function(req, res) {  
  res.send('ver pagina de inicio');  
});
```



# Express > Routing

## ■ Métodos de respuesta (Response)

- Hay que indicar uno de estos métodos para que una solicitud no quede pendiente.

Método	Descripción
<a href="#"><u>res.send()</u></a>	Envía una respuesta en forma de cadena
<a href="#"><u>res.json()</u></a>	Envía una respuesta en formato JSON.
<a href="#"><u>res.redirect()</u></a>	Redirecciona a otra URL. Ej redireccionar a "/home" <pre>res.redirect("/home");</pre>
<a href="#"><u>res.render()</u></a>	Renderiza una plantilla y envía la renderización como respuesta.
<a href="#"><u>res.sendFile()</u></a>	Envía un archivo como una secuencia de octetos.
<a href="#"><u>res.sendStatus()</u></a>	Establece el código de estado de la respuesta y envía su representación de string como el cuerpo de respuesta.
Otros...	

# Express > Routing

## ■ Organización de rutas en módulos

- Las aplicaciones deben optar por un **diseño modular**, se mejora la arquitectura y la reutilización.
- Cada módulo se debería encargar de gestionar las rutas de una entidad.
  - Index, usuarios, canciones, etc.
- La carpeta **routes** se utiliza comúnmente para almacenar los módulos de rutas
- En cierto modo estos módulos hacen el papel de **controladores**

```
Application
├── app.js
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
├── index.js
├── users.js
└── views
    ├── error.html
    └── index.html
```

# Express > Routing

## ■ Organización de rutas en módulos

- Para declarar un módulo se utiliza *module.exports*
- Un módulo puede recibir parámetros en su constructor

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

- Para incluir el módulo en la aplicación usamos *require(fichero)* (*parámetros*)

```
var app = express();  
//Rutas/controladores por lógica  
require("./routes/users.js")(app); // (app, param1, param2, etc.)  
require("./routes/songs.js")(app); // (app, param1, param2, etc.)  
... .
```

# Express > Peticiones web

## ■ Peticiones GET y parámetros

- Las peticiones GET pueden contener parámetros en su URL.
- Los parámetros se pueden enviar de dos formas:
  1. Enviando la **clave y el valor** como elementos de la URL usando los caracteres **? y &** para concatenar. Ejemplos:
    - `http://localhost/canciones?nombre=despacito`
    - `http://localhost/canciones?nombre=despacito&autor=Luis Fonsi`
  2. **Embebiendo el valor del parámetro** en la URL sin especificar la clave.
    - `http://localhost/cancion/234/`

# Express > Peticiones web

## ■ Obteniendo parámetros Get

- Se utiliza el objeto *query* incluido en la petición (req)
  - *req.query.<clave\_parámetro>*
  - Si el parámetros no existe, la petición retornará "undefined"
    - Deberíamos comprobar si el parámetro es *null* o *undefined*
    - Para comprobar si el parámetro es "undefined" usamos la función *typeof()*

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

# Express > Peticiones web

## ■ Obteniendo parámetros Get

- Todos los valores que obtenemos a través del `req.query` son *cadenas de texto*.
  - Sí queremos otro tipo de datos debemos convertirlos
  - Ej, para convertir en enteros usaríamos: `parseInt(req.query.num1)`
- JavaScript define varias funciones para cambiar el tipo de las variables, ejemplos:
  - `parseInt("valor")`
  - `parseFloat("valor")`

```
parseFloat("3.14");  
parseFloat("314e-2");  
parseFloat("0.0314E+2");
```

# Express > Peticiones web

## ■ Obteniendo parámetros en URLs

- Ejemplo: `http://localhost:8081/canciones/121/`
- La URL debe especificar la posición del parámetro
  - `:<clave_parámetro>`
- Se utiliza el objeto ***params*** definido en el objeto petición (req)
  - `req.params.<clave_parámetro>`

```
app.get('/canciones/:id', function(req, res) {  
  var respuesta = 'id: ' + req.params.id;  
  res.send(respuesta);  
})
```

# Express > Peticiones web

## ■ Peticiones POST y parámetros

- Se utiliza comúnmente para el envío de información a través de *formularios*.
- A diferencia de las peticiones GET, las POST *tienen un cuerpo (body)* que puede contener datos:
  - Pares de clave-valor (parámetros), texto plano, json, binario, etc.



# Express > Peticiones web

## ■ Enviando parámetros POST

- Los parámetros de una petición POST se **envían en el cuerpo** del mensaje (body).
- Ej, formulario que envía una petición **POST /cancion**.
- Define inputs con atributo **name** .
  - El **name** será la clave del parámetros, por ejemplo: nombre, genero, precio, etc.

```
<form method="post" action="/cancion">  
  <input type="number" name="precio" />  
  ...  
</form>
```

A screenshot of a web form titled "Agregar canción". The form contains the following fields and controls:

- Nombre:** A text input field with the value "nuev".
- Genero:** A dropdown menu with "Pop" selected.
- Precio (€):** A text input field with the value "2".
- Imagen portada:** A button labeled "Seleccionar archivo" and the text "Ningún archivo seleccionado".
- Fichero audio:** A button labeled "Seleccionar archivo" and the text "Ningún archivo seleccionado".
- Submit:** A blue button labeled "Agregar".

# Express > Peticiones web

## ■ Obteniendo los parámetros por POST

- Para acceder a los parámetros incluidos en el body necesitamos añadir un módulo externo, como **body-parser**, (aunque hay otros) se instala mediante comando:

```
npm install body-parser --save
```

- Implementación: se instancia el módulo
  - Agregamos los módulos en el fichero **app.js**
  - Se obtiene el objeto **body-parser** con el **require(módulo)**
  - Se agregan funciones de parseo a la aplicación con **app.use()**
    - **Urlencoded** parsea cuerpos en formato URL, pares clave-valor (estándar formularios) El extendido permite procesar valores como objetos ricos JSON
    - **Json()** parsea cuerpos en formato JSON (usado por muchos Servicios Web)

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

# Express > Peticiones web

## ■ Obteniendo los parámetros por POST

- Para acceder a los parámetros del body de la petición usamos el objeto *body* incluido en el objeto petición (req).
  - *req.body.<clave\_parámetro>*
- Al igual que en parámetros anteriores estos podrían tomar valores *undefined* o *null*.

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

# Express > Recursos estáticos

- Express provee una función de asistencia (middleware) para facilitar el acceso a **recursos estáticos**.
  - Imágenes, videos, scripts, css, etc.
- Por convenio, **public** es el directorio donde se almacenan los ficheros estáticos.
- Estos ficheros son servidos por la aplicación sin pasar por ningún controlador

```
Application
├─ app.js
├─ package.json
├─ public
│   └─ images
│   └─ javascripts
│   └─ stylesheets
│       └─ style.css
├─ routes
│   └─ index.js
│   └─ users.js
└─ views
    └─ error.html
    └─ index.html
```

# Express > Recursos estáticos

- **Caso 1:** Declarar un directorio estático estándar usando la función:
  - `express.static(<ruta del directorio>)`.
  - Los ficheros se obtienen desde la raíz del sitio `/`
- **Caso 2:** Declarar un directorio estático con una **vía acceso virtual** (donde la ruta NO existe realmente en el directorio de archivos), se tiene que crear un alias o prefijo.
  - `express.static('alias', <ruta del directorio>)`.

```
├─ public
│   └─ images
│       └─ stylesheets
```

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

// Caso 1.
app.use(express.static('public'));

// acceso.
http://localhost/images/user.png
http://localhost/stylesheets/style.css

// Caso 2.
app.use('static', express.static('public'));

// acceso.
http://localhost/static/images/user.png
http://localhost/static/stylesheets/style.css
```

# Express > Vistas y plantillas

- Una de las formas más comunes para intercalar datos relativos a la lógica de negocio en vistas HTML son las **plantillas**.
- Los **motores de plantillas** tienen **acceso a los atributos del modelo, pudiendo:**
  1. Insertar los atributos en el HTML
  2. Usarlos en estructuras de control y utilidades (condiciones, bucles, etc.)

```
Application
├─ app.js
├─ package.json
├─ public
│   └─ images
│   └─ javascripts
│   └─ stylesheets
│       └─ style.css
├─ routes
│   └─ index.js
│   └─ users.js
└─ views
    └─ error.html
    └─ index.html
```

# Express > Vistas y plantillas

- Hay muchos motores de plantillas que se pueden utilizar en Node:
  - Swig, Pug, Moustache, EJS, swig, etc.
- Comúnmente las plantillas se almacenan en el directorio **views**
- La mayor parte de motores de plantillas deben ser instalados, nosotros utilizaremos **swig**
  - Instalación swig:

```
npm install swig --save
```

```
Application
├── app.js
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.html
    └── index.html
```

# Express > Vistas y plantillas

## ■ Definición de plantillas

- Los atributos del modelo se referencian con:  
**{{ <nombre\_del\_atributo> }}**
- Ejemplo: al renderizar la plantilla se insertará el valor del atributo **vendedor**

```
<html>
  <head>
    <title>Canciones</title>
  </head>
  <body>
    <h1> {{ vendedor }} </h1>
  </body>
</html>
```

Plantilla swig

```
{ "vendedor" : "uniovi" }
```

Atributos del modelo



# Express > Vistas y plantillas

## ■ Definición de plantillas

- Los valores de los atributos pueden ser también tipos **objetos**
- El operador `.` permite acceder a los atributos

```
<html>
  <head>
    <title>Canciones</title>
  </head>
  <body>
    <h1>{{ vendedor.nombre }}</h1>
  </body>
</html>
```

Plantilla swig

```
{ "vendedor" :
  {
    "nombre" : "uniovi",
    "año" : 2000
  }
}
```

Atributos del modelo


# Express > Vistas y plantillas

## ■ Definición de plantillas

- Los valores de los atributos pueden ser **colecciones**
- Ofrece **estructuras de control** para recorrer las colecciones
  - {% for <variable temporal> in <colección atributo del modelo> %} {% endfor %}

```
<body>
<ul>
  {% for cancion in canciones %}
    <li>
      {{ cancion.nombre }} -
      {{ cancion.precio }}
    </li>
  {% endfor %}
</ul>
</body>
```

Plantilla swig



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

Atributos del modelo

# Express > Vistas y plantillas

## ■ Definición de plantillas

- Ofrece **estructuras condicionales** para incluir código en base a una expresión lógica
  - `{% if <expresión lógica> %} {% endif %}`

```
<body>  
  {% if coche.precio < 100 %}  
    <p> oferta </p>  
  {% endif %}  
</body>
```

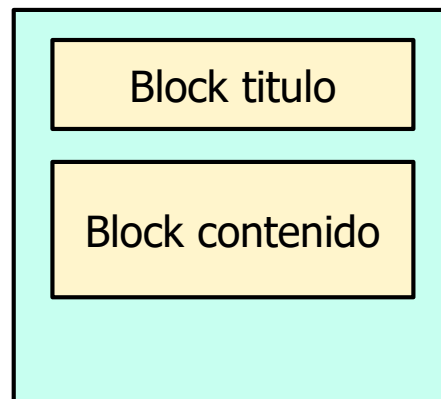
- Soporta casi el mismo conjunto de expresiones lógicas definido en Javascript
- Documentación completa:

<http://node-swig.github.io/swig-templates/docs/tags/#if>

# Express > Vistas y plantillas

## ■ Definición de plantillas

- Ofrece un sistema de **composición de plantillas** basado en **herencia** y redefinición de **bloques**
- Se define una **plantilla base** ej: base.html con todos los elementos comunes a todas las vistas
  - Se divide de forma lógica el código de la plantilla base bloques que podrán ser redefinidos en sus hijos. (Ej bloques: cabeceras, titulo, contenido, etc)
    - `{% block <nombre> %}` contenido redefinible `{% endblock %}`

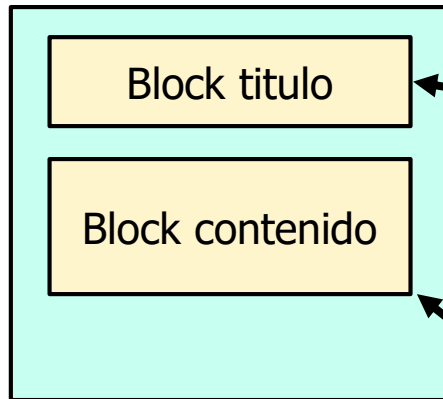


Base.html

# Express > Vistas y plantillas

## ■ Definición de plantillas

- Ejemplo, declaración de bloques en plantilla base.html



Base.html

```
<html lang="en">
<head>
  <title>
    {% block titulo %} uoMusic {% endblock %}</title>
  <link rel="stylesheet" href="min.css"/>
</head>
<body>

  <div class="container">
    <!-- Contenido -->
    {% block contenido %}
      <p> Contenido redefinible </p>
    {% endblock %}
  </div>
```

# Express > Vistas y plantillas

## ■ Definición de plantillas

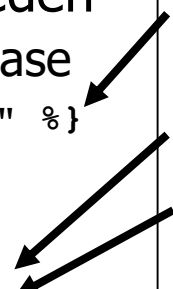
- El resto de plantillas pueden extender de una plantilla base

```
{% extends "<path_plantilla>" %}
```

- Pueden redefinir o no el contenido de los bloques definidos en la plantilla base

```
{% block nombre %}  
    nuevo contenido  
{% endblock %}
```

```
{% extends "base.html" %}  
  
{% block titulo %}Tienda{% endblock %}  
  
{% block contenido %}  
<h2>Canciones</h2>  
<ul>  
    {% for cancion in canciones %}  
        <li>{{ cancion.nombre }} </li>  
    {% endfor %}  
</ul>  
{% endblock %}
```



# Express > Vistas y plantillas

## ■ Renderización de plantillas en la aplicación

- Se instancia el objeto **swig**
  - Normalmente se instancia en **app.js** junto al resto de módulos globales

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

- Desde **app.js** se envía en el constructor de los módulos controladores, son los que van a necesitar renderizar plantillas

app.js

```
require("./routes/rusuarios.js")(app, swig);
```

rusuarios.js

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

# Express > Vistas y plantillas

## ■ Renderización de plantillas en la aplicación

- El objeto **swig** contiene una función **renderFile(<plantilla>,<modelo de datos>)** que retorna el código generado.
  - El modelo de datos es un objeto con pares **clave : valor**.
  - Los valores pueden ser tipos simples, objetos, o colecciones
  - El código generado por el render suele ser retornado como respuesta

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

```
<h1> {{ vendedor }} </h1>  
  
{% for cancion in canciones %}  
  <li>{{ cancion.nombre }} </li>  
{% endfor %}
```



# Sistemas Distribuidos e Internet

## Tema 7

### Introducción a Node JS

