



Escuela de Ingeniería Informática

Escuela de Ingeniería Informática  
School of Computer Science Engineering

Universidad de Oviedo  
*Universidá d'Uviéu*  
*University of Oviedo*

# Sistemas Distribuidos e Internet

## Tema 7 Introducción a Node.js



**Dr. Edward Rolando Núñez Valdez**

nunezedward@uniovi.es

# 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

# ¿Qué es Node.js?

- Es una plataforma de software que permite ***ejecutar JavaScript del lado del servidor***
- Características principales:
  - Arquitectura basada en eventos - (***Single Threaded Event Loop***).
    - Utiliza un único hilo de ejecución que gestiona las peticiones concurrentes de los clientes.
  - Modelo de gestión de operaciones de ***I/O se forma asíncrona y sin bloqueo*** (Non-blocking)
    - Permite ejecutar varios procesos de E/S de forma simultánea sin producir un bloqueo en el sistema.
  - Mediante su ***diseño modular*** que permite construir aplicaciones web de alto rendimiento y escalables.



# ¿Qué es Node.js?

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



# ¿Qué 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 de programación
  - 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.
  - Es coherente con la tendencia actual de Cloud Technology



# 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.



# Características y Arquitectura de Node.js



# Características y Arquitectura de Node.js

- Basado en un diseño modular.
- Modelo de operaciones I/O asíncrona o sin bloqueo (Non-blocking).
- Arquitectura basada en eventos (Single Threaded Event Loop).



# 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.js se realiza con **NPM (Node Package Manager)** o Yarn.

# Principales características de Node.js

- **NPM (Node Package Manager)**

- Es el **gestor de paquetes** para JavaScript y Node.js.
- 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)**

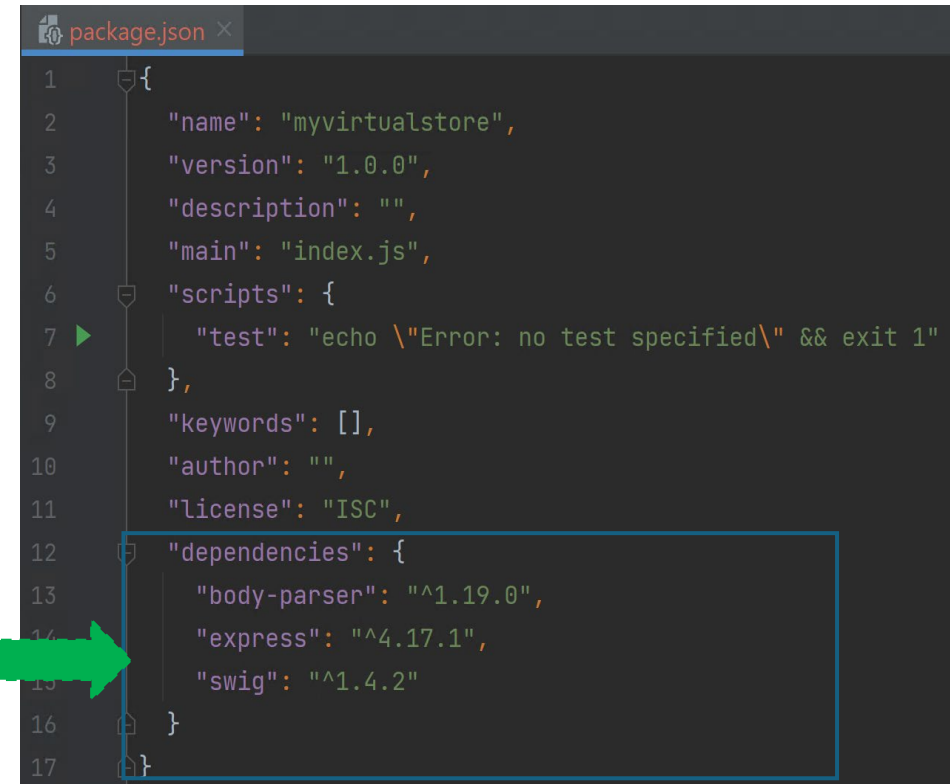
- Los módulos se descargan e instalan **localmente** mediante el comando
  - Esto creará el directorio **node\_modules** en el directorio actual (si no existe) y descargará el paquete a ese directorio.

```
npm install <package_name>
```



# Principales características de Node.js

- **Declarar el uso de un 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.



```
1 {
2   "name": "myvirtualstore",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1"
8   },
9   "keywords": [],
10  "author": "",
11  "license": "ISC",
12  "dependencies": {
13    "body-parser": "^1.19.0",
14    "express": "^4.17.1",
15    "swig": "^1.4.2"
16  }
17 }
```

# 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:
- Ejemplo:

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

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





# Principales características de Node.js

- Algunos comandos importantes de npm

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



# 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.



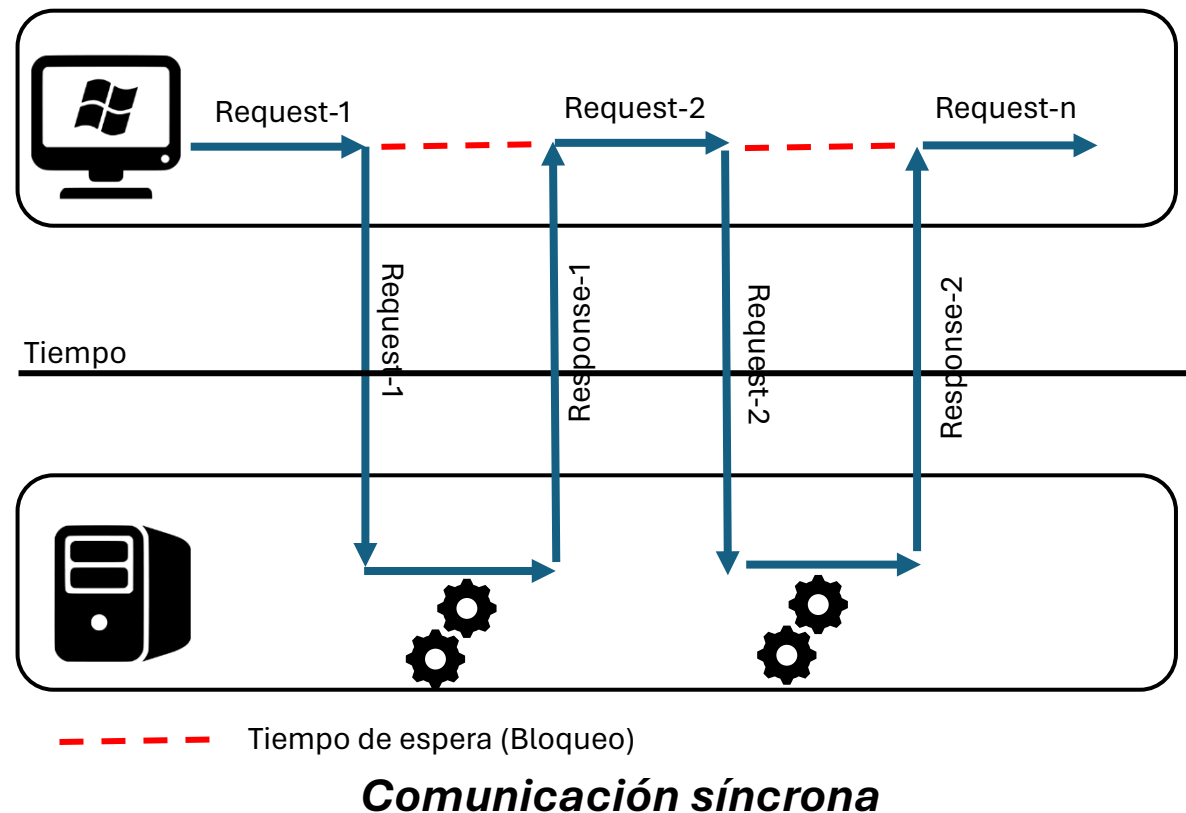
# Principales características de Node.js

- **Características de una comunicación *síncrona***
  - Las operaciones se ejecutan de forma secuencial.
  - Las operaciones son bloqueantes (blocking).
    - El programa permanece bloqueado hasta que termine la operación.
  - El cliente espera la respuesta del servidor para continuar con el flujo del programa.
  - Son menos eficientes, pero más simples y fiables que las operaciones asíncronas.



# Principales características de Node.js

- **Funcionamiento 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;
}

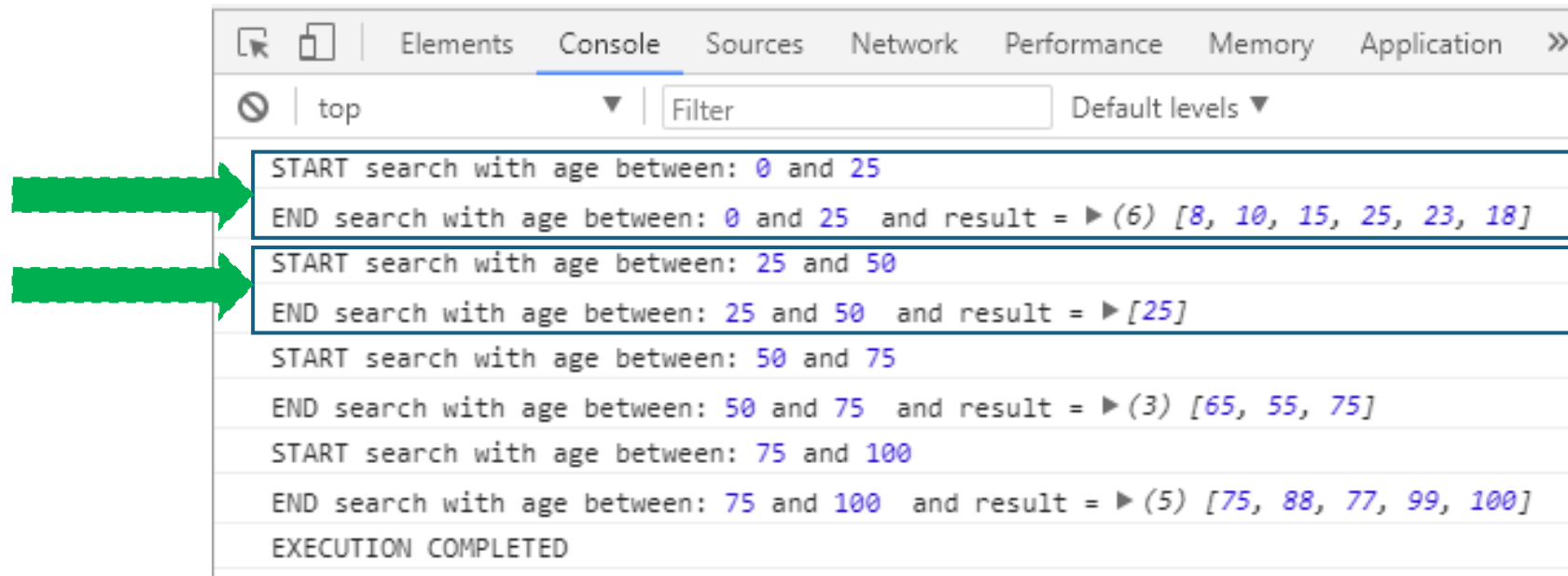
function executeSearch() {
  var maxAge = 0;
  var maxRange = 4;
  for (var i = 0; i < maxRange; i++) {
    minAge = maxAge;
    maxAge = maxAge + 25;
    result = Search(minAge, maxAge);
    console.log('END search with age between:', minAge, "and",
      maxAge, ' and result =', result);
  };

  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
```



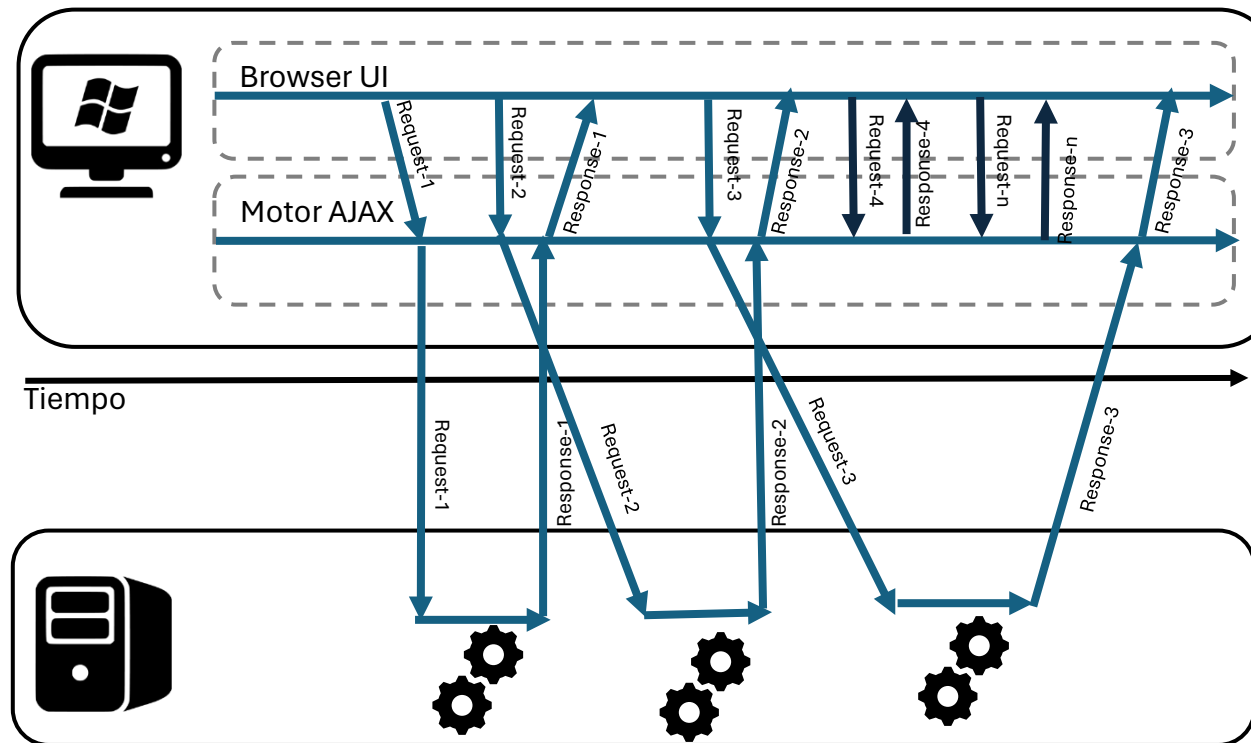
# Principales características de Node.js

- **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), promesas o Async/await*.
  - 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);
}

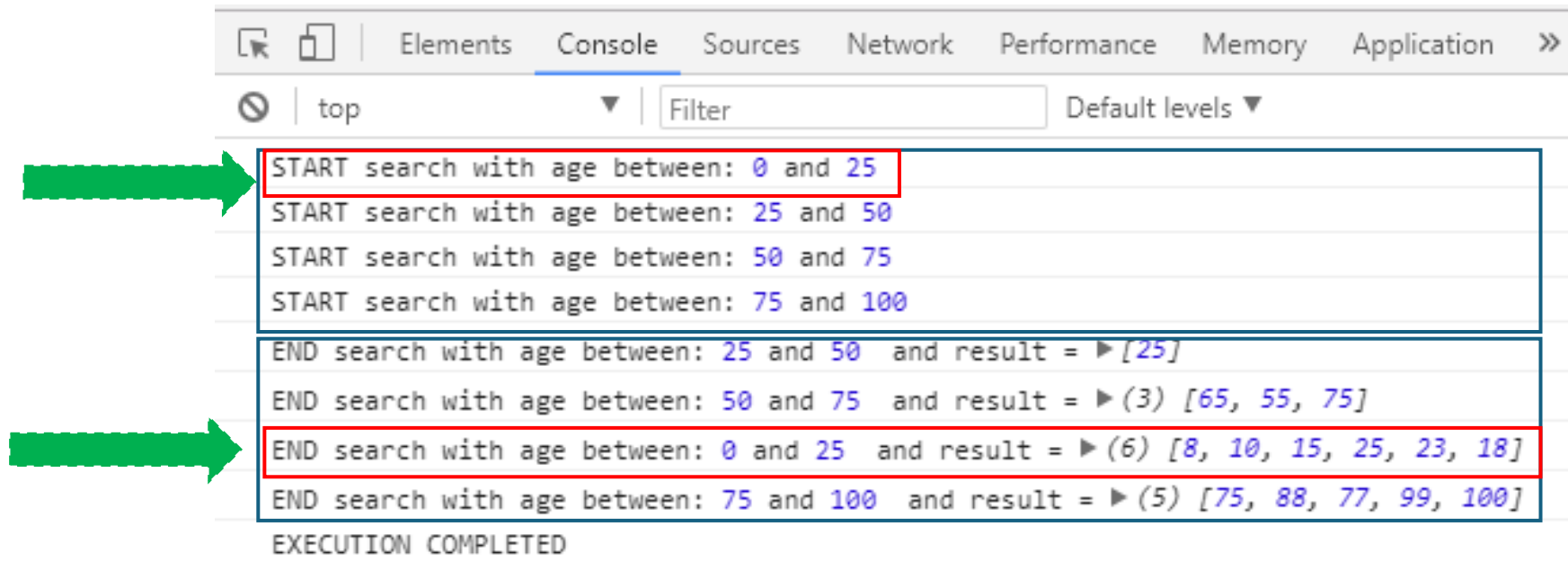
function executeAsyncSearch() {
  var maxAge = 0;
  var maxRange = 4;
  var count = 0;
  for (var i = 0; i < maxRange; i++) {
    minAge = maxAge;
    var maxAge = maxAge + 25;
    asyncSearch(minAge, maxAge, function (minAge, maxAge, result) {
      console.log('END search with age between:', minAge,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



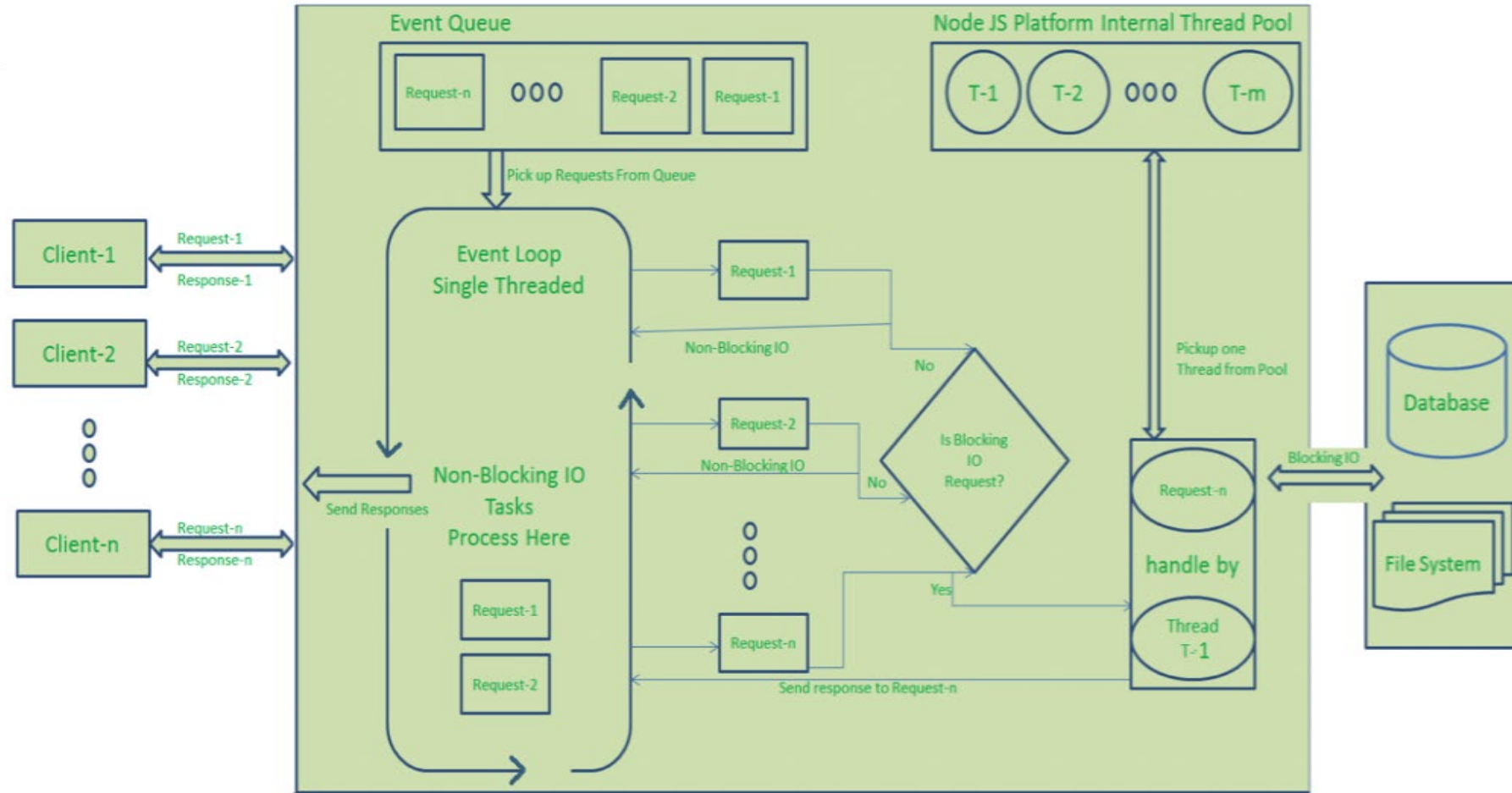
# Principales características de Node.js

- **Arquitectura basada en eventos**

- Node.js 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



Node JS Application/Node JS Server  
Fuente: <https://www.journaldev.com>

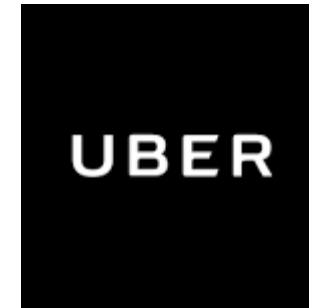


# ¿Cuándo usar o No Node.js?

- 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?



GitHub



# Entorno de desarrollo

# Entorno de desarrollo

- Requisitos:
  - Descargar e instalar Node.js en el sistema operativo.
- Pueden desarrollarse aplicaciones prácticamente en cualquier IDE
  - Notepad, Visual Studio Code, Eclipse, *IntelliJ IDEA*, *WebStorm*, Spring Tool Suite, etc.
  - Plugins requeridos: *JavaScript and TypeScript*, *Node.js* (depende del IDE)





# Entorno de desarrollo

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

LTS  
Recommended For Most Users

Current  
Latest Features

  
Windows Installer  
node-v20.11.1-x64.msi

  
macOS Installer  
node-v20.11.1.pkg

  
Source Code  
node-v20.11.1.tar.gz

Windows Installer (.msi)

Windows Binary (.zip)

macOS Installer (.pkg)

macOS Binary (.tar.gz)

Linux Binaries (x64)

Linux Binaries (ARM)

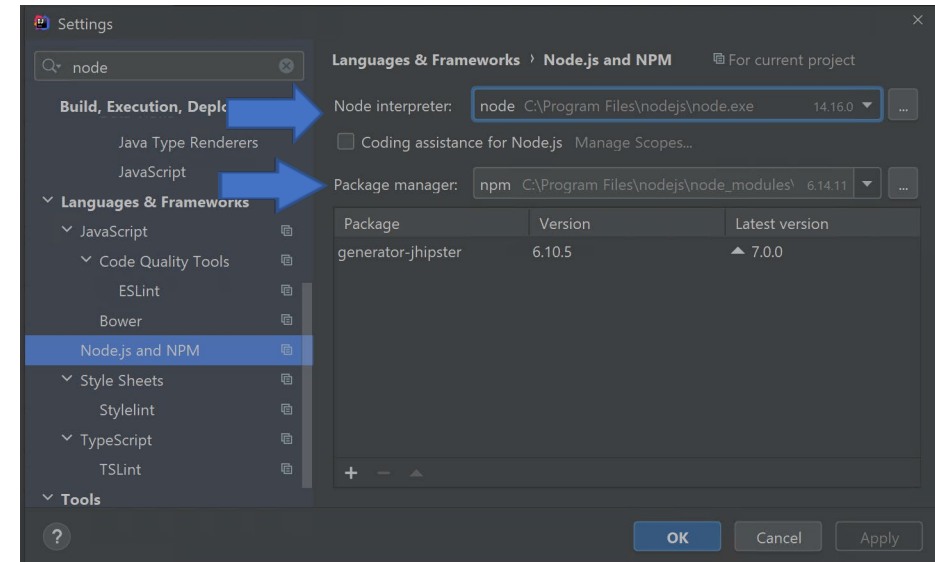
Source Code

32-bit	64-bit	ARM64
32-bit	64-bit	ARM64
64-bit / ARM64		
64-bit	ARM64	
64-bit		
ARMv7	ARMv8	
node-v20.11.1.tar.gz		



# Entorno de desarrollo > IntelliJ IDEA

- En IntelliJ IDEA Ultimate los plugins vienen instalados y habilitados por defecto.
- Con IntelliJ IDEA, puede tener varias instalaciones de Node.js y alternar entre ellas mientras trabaja en el mismo proyecto
- Node Version Manager (NVM)
  - <https://github.com/nvm-sh/nvm/blob/master/README.md>

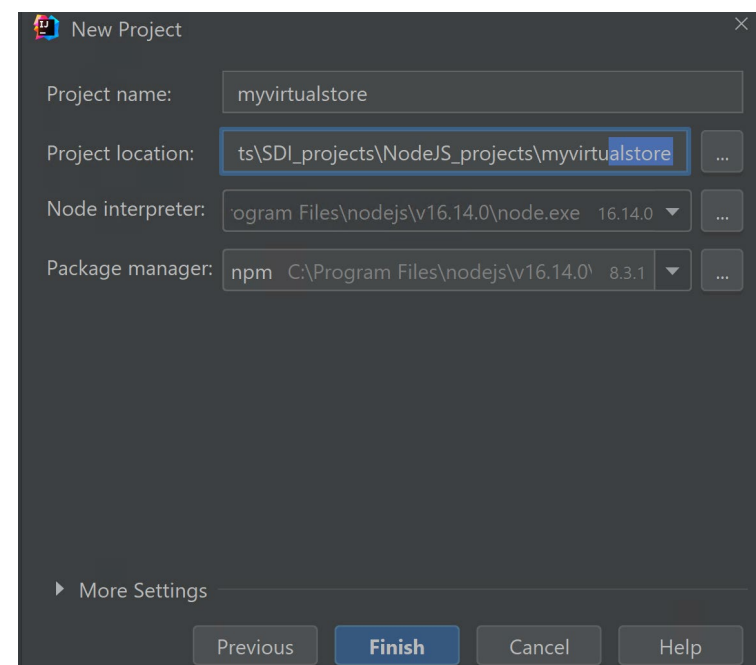
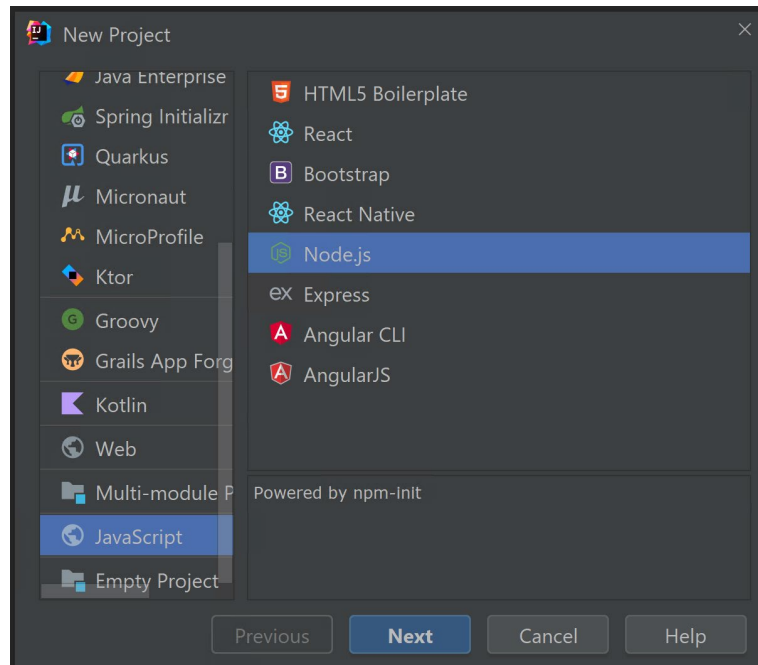


```
$ nvm use 16
Now using node v16.9.1 (npm v7.21.1)
$ node -v
v16.9.1
$ nvm use 14
Now using node v14.18.0 (npm v6.14.15)
```



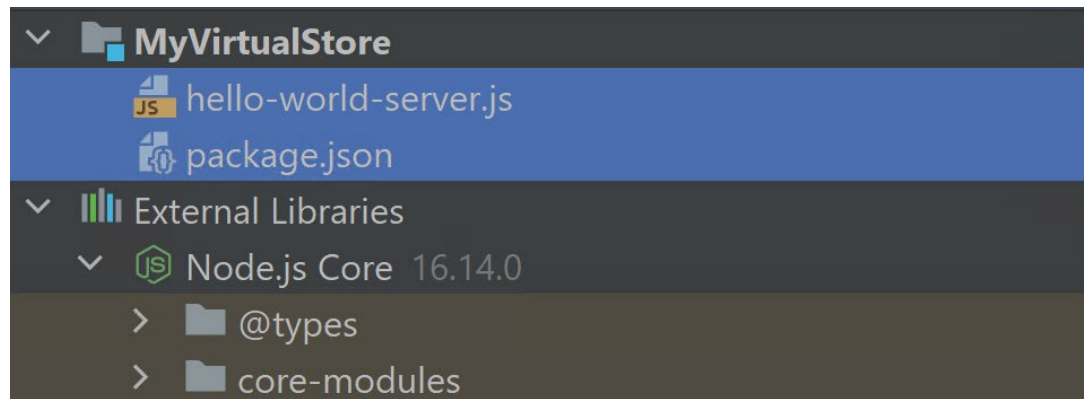
# Aplicación

- Ejemplo creación de una aplicación web (hola mundo) Node.js en IntelliJ IDEA
  - Menú **File | New | Project**



# Aplicación > Estructura básica

- Los ficheros necesarios:
  - Fichero ***package.json*** define la **configuración y los metadatos** de la aplicación (viene por defecto)
  - Añadimos un fichero con la **lógica del negocio** (***hello-world-server.js***)



# Aplicación > Server

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

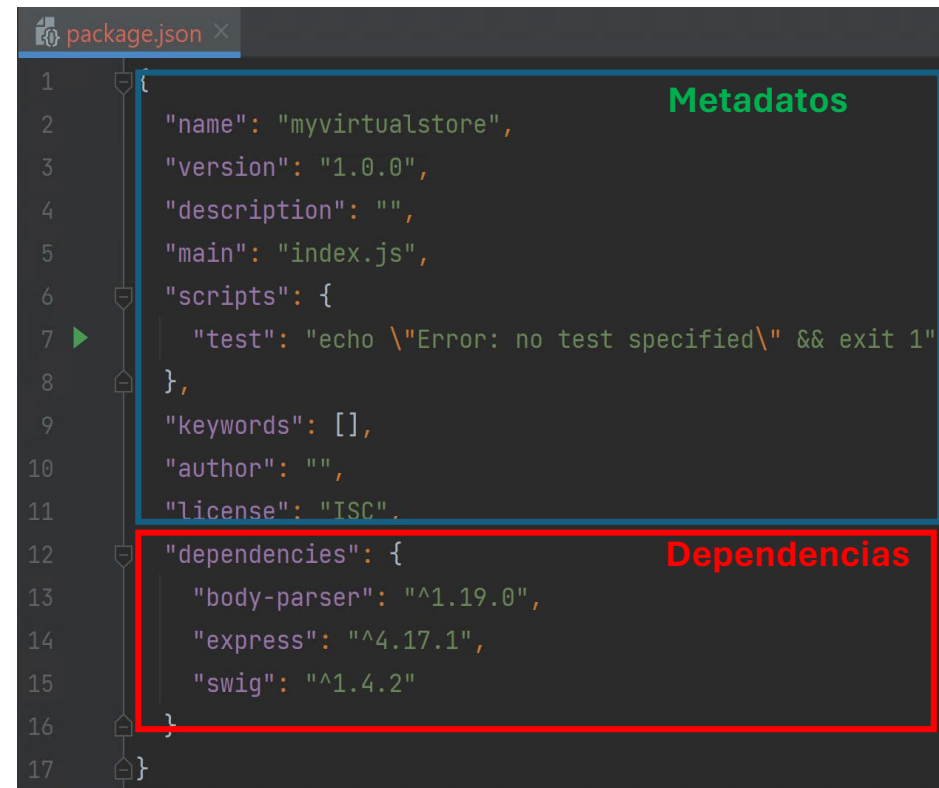
- Contiene la *lógica de la aplicación*
- Inicialmente se define una variable que incluye el **módulo http** que viene incluido con Node.js
- Este módulo permite crear aplicaciones web muy simples
  - En lugar de este módulo usaremos **express**

```
JS hello-world-server.js x
1 let http = require('http');
2 http.createServer(function handler(req : IncomingMessage , res : ServerResponse ) {
3     res.writeHead( statusCode: 200, headers: {'Content-Type': 'text/plain'});
4     res.end( chunk: 'Hello World\n');
5 }).listen( port: 1337, hostname: '127.0.0.1');
6 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 o yarn**
  - Por defecto se añaden a la lista de “dependencies”

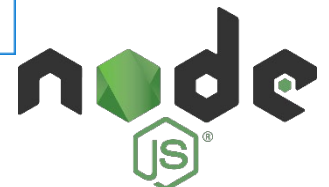
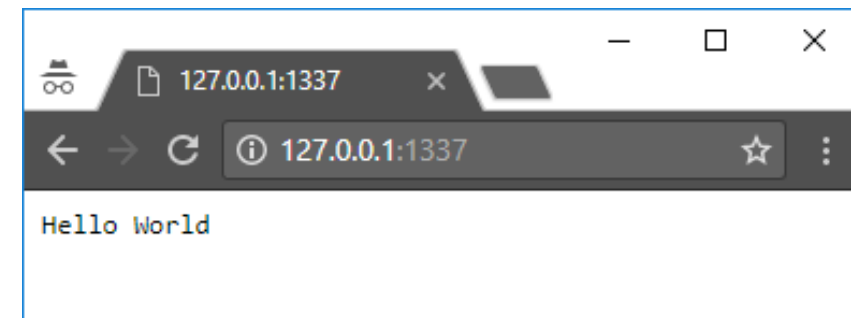
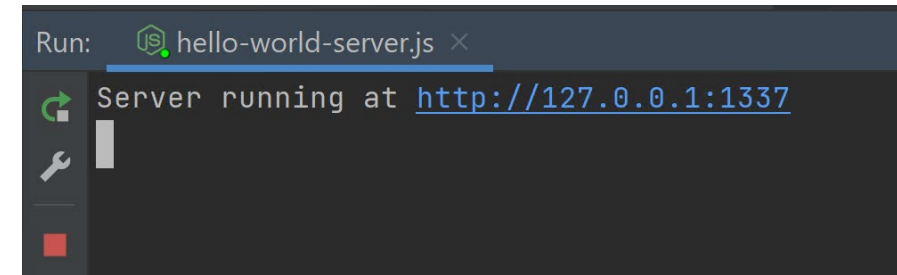
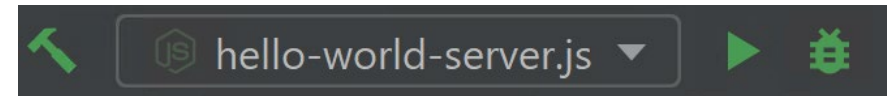


```
1 {  
2   "name": "myvirtualstore",  
3   "version": "1.0.0",  
4   "description": "",  
5   "main": "index.js",  
6   "scripts": {  
7     "test": "echo \"Error: no test specified\" && exit 1"  
8   },  
9   "keywords": [],  
10  "author": "",  
11  "license": "ISC",  
12  "dependencies": {  
13    "body-parser": "^1.19.0",  
14    "express": "^4.17.1",  
15    "swig": "^1.4.2"  
16  }  
17 }
```

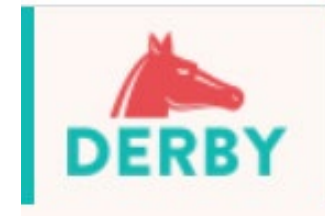


# Aplicación > Despliegue

- Clic derecho en el fichero `hello-world-server.js` | **Run *hello-word-server.js...***
- 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



# Frameworks para Node.js





# 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 métodos (Get, Post, Put, Delete, etc.)
  - Permite trabajar con distintos motores de plantillas (Jade, Swig, TWIG, 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 > Generador Apps

- express-generator
  - Herramienta que permite crear rápidamente un esqueleto de aplicación Node.js-express.
    - Puede ejecutar el generador de aplicaciones con el comando npx (disponible en Node.js 8.2.0).

```
npx express-generator --view=twig myapp2
```

- O instalar el módulo si tenemos una versión de Node.js anterior a la 8.2.0

```
npm install -g express-generator  
express --view= twig myapp2
```

- Instalar dependencia y lanzar aplicación

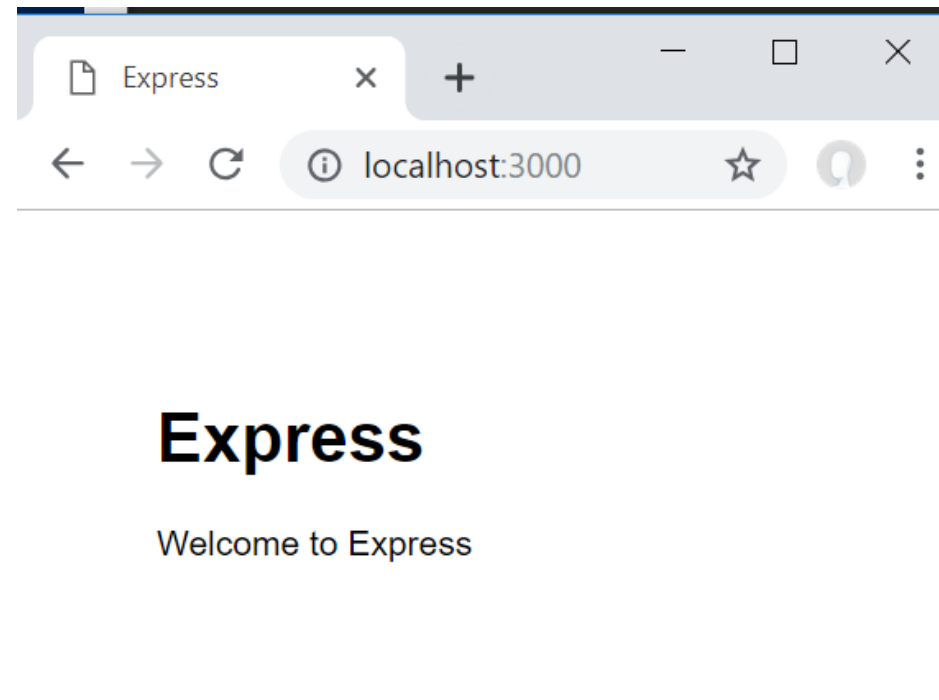
```
$ cd myapp2  
$ npm install  
$ npm start
```



# Express > Generator Apps

- Resultado -> express-generator

```
create : myapp2\  
create : myapp2\public\  
create : myapp2\public\javascripts\  
create : myapp2\public\images\  
create : myapp2\public\stylesheets\  
create : myapp2\public\stylesheets\style.css  
create : myapp2\routes\  
create : myapp2\routes\index.js  
create : myapp2\routes\users.js  
create : myapp2\views\  
create : myapp2\views\error.pug  
create : myapp2\views\index.pug  
create : myapp2\views\layout.pug  
create : myapp2\app.js  
create : myapp2\package.json  
create : myapp2\bin\  
create : myapp2\bin\www
```



# Express > Estructura Aplicación

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

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



# Express > Aplicación

- **App.js**

- Fichero principal de la aplicación.
- La función **require** indica que se utilizará un **módulo ...**
- La función **express()** crea una **nueva aplicación express**.
- La aplicación express se configura por medio de funciones que responden a **métodos http**.

Application

└─ app.js

...

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

let indexRouter = require('./routes/index');
let usersRouter = require('./routes/users');

let app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'twig')

...

app.get('/songs', function (req, res) {
  res.send('lista de canciones');
});
```



# Express > Aplicación

- **Carpeta bin**

- Es una carpeta genérica
- Sirve para guardar ficheros que contengan funciones, objetos o variables de uso genérico.
- Por ejemplo, creación del servidor HTTP, puertos por los que escucha el servidor, etc.

Application

```
|— bin  
|   |— www.js  
... •
```

```
let app = require('../app');  
  
let http = require('http');  
  
let port = normalizePort(process.env.PORT || '3000');  
  
app.set('port', port);  
  
let server = http.createServer(app);  
  
server.listen(port);  
server.on('listening', onListening);  
...
```



# Express > Aplicación

- **Uso de variables de aplicación**

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

Application

├─ bin

| └─ **www.js**

...

```
...  
let port = normalizePort(process.env.PORT || '3000');  
app.set('port', port);  
  
let server = http.createServer(app);  
  
//server.listen(port);  
server.listen(app.get('port'));  
...
```





# Express > Routing

- **Routing (enrutamiento o direccionamiento)**

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

- La definición de ruta tiene la siguiente estructura:

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

Donde:

- **app**: es una instancia de una aplicación express.
- **METHOD**: es método de ruta que se corresponde un método de solicitud HTTP.
- **PATH**: es una vía de acceso al servidor.
- **HANDLER**: es una función, matriz de funciones o un conjunto de ambas que se ejecuta cuando se realiza una petición la ruta.

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

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

..
```



# Express > Routing

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

- **Method (Método):**
  - Se corresponde con un método de solicitud HTTP.
  - Soporta múltiples métodos HTTP
    - GET, POST, PUT, DELETE, etc.

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

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

app.post('/songs/add', function (req, res) {
  res.send('lista de canciones');
});

app.put('/songs/:id', function (req, res) {
  res.send('lista de canciones');
});

..
```



# Express > Routing

*app.METHOD(PATH, HANDLER)*

- 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 **cadena**s
    - 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 “/song” a las peticiones GET, POST o PUT

```
let express = require('express');
let app = express();
...

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

app.get('/song/:id', function (req, res) {
  res.send('Info canción');
});

app.post('/song', function (req, res) {
  res.send('lista de canciones');
});

app.put('/song', function (req, res) {
  res.send('lista de canciones');
});
..
```



# Express > Routing

- **Enrutamiento y comodines**

- En las rutas se admite el uso de comodines, como por ejemplo, ?, +, \*, () y otras expresiones regulares
- Ejemplos basados en patrones de cadenas o series
  - Ejemplo: cualquier ruta que **comience con promo:**

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

```
// /proclamar, /proMOCIONar /procesar
app.get('/pro*ar', function (req, res) {
  res.send('Respuesta al patrón pro*ar');
});
```

```
// ruta que coincide con abcd, abbcd, abbbcd, etc.
app.get('/ab+cd', function (req, res) {
  res.send('Respuesta al patrón ab+cd ');
});
```

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



# Express > Routing

`app.METHOD(PATH, HANDLER)`

- Manejadores (Handler)
  - **Función, matriz de funciones o un conjunto** de ambas que se ejecutan 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('/songs', function (req, res) {  
  res.send('lista de canciones');  
});
```

```
let authenticator = function (req, res, next) {  
  console.log('ejecutar un autenticador');  
  next();  
}  
  
let validator = function (req, res, next) {  
  console.log('Por ejemplo, lógica de validación de datos');  
  datos_validos = true;  
  if (datos_validos == true) {  
    next();  
  } else {  
    res.send("Datos no válidos")  
  }  
}  
  
app.post('/songs', [authenticator, validator], function (req, res, next) {  
  console.log('Ejecutar aquí alguna logica de negocio');  
  next();  
}, function (req, res) {  
  res.send('Ver info canción');  
});
```

# 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.post('/song', function (req, res) {  
  res.send('lista de canciones');  
});
```



# Express > Routing

- **Métodos de respuesta (Objeto Response)**

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

Método	Descripción
<code>res.send()</code>	Envía una respuesta en forma de cadena
<code>res.json()</code>	Envía una respuesta en formato JSON.
<code>res.redirect()</code>	Redirecciona a otra URL. Ej redireccionar a “/home” <pre>res.redirect("/home");</pre>
<code>res.render()</code>	Renderiza una plantilla y envía la renderización como respuesta.
<code>res.sendFile()</code>	Envía un archivo como una secuencia de octetos.
<code>res.sendStatus()</code>	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
├─ bin
│   └─ www.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
- Para incluir el módulo en la aplicación usamos ***require(fichero) (parámetros)***

```
module.exports = function (app) {  
  app.get('/songs', function (req, res) {  
    res.send('lista de canciones');  
  });  
}
```

```
// Dos formas distintas  
require('./routes/songs')(app); // (app, param1, param2, etc.)  
let usersRouter = require('./routes/users');  
app.use('/users', usersRouter);
```



# Express > Routing

- **app.route()**

- Permite crear manejadores de *rutas encadenables*.
- La *vía de acceso* se especifica en una *única ubicación*.
- Ayuda a reducir la redundancia y errores tipográficos.

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

```
module.exports = function (app) {  
  app.route('/songs')  
    .get(function (req, res) {  
      res.send("ver canciones");  
    })  
    .post(function (req, res) {  
      res.send("crear una canción");  
    })  
    .put(function (req, res) {  
      res.send("actualizar una canción");  
    })  
};
```

# Express > Peticiones web

- **Peticiones HTTP GET y parámetros**

- Las peticiones HTTP 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/songs?name=despacito`
    - `http://localhost/songs?name=despacito&autor=Luis Fonsi`
  2. **Embebiendo el valor del parámetro** en la URL sin especificar la clave.
    - `http://localhost/song/234/`



# Express > Peticiones web

## • Obteniendo parámetros HTTP GET

- Se utiliza el objeto **query** incluido en la petición (req)
  - **req.query.<clave\_parámetro>**
- Si el parámetro 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()**

```
module.exports = function (app) {  
  app.route('/songs')  
    .get(function (req, res) {  
      let response = "";  
      if (req.query.name != null)  
        response += 'Name: ' + req.query.name;  
      if (typeof req.query.author != "undefined")  
        response += 'Author: ' + req.query.author;  
      res.send(response);  
    })  
    ...  
};
```



# Express > Peticiones web

## • Obteniendo parámetros HTTP GET

- Todos los valores que obtenemos a través del **req.query son cadenas de texto.**
  - Sí queremos otro tipo de datos debemos convertirlos
  - Por ejemplo, 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 embebidos/incrustado en la URL**

- Ejemplo: `http://localhost:8081/songs/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('/songs/:id', function (req, res) {  
  let response = 'Id: ' + req.params.id;  
  res.send(response);  
});
```



# Express > Peticiones web

- **Peticiones HTTP POST y parámetros**

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



# Express > Peticiones web

- **Enviando parámetros por HTTP POST**

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

```
<form method="post" action="/songs">  
  <input type="text" name="title" />  
  <input type="text" name="kind" />  
  <input type="number" name="price" />  
  
  ...  
</form>
```

Agregar canción

Nombre:

Genero:

Precio (€):

Imagen portada:  
 Ningún archivo seleccionado

Fichero audio:  
 Ningún archivo seleccionado





# 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
```

- 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)

```
let app = express();  
let 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('/songs', function (req, res) {  
  res.send("Canción agregada:" + req.body.title + "<br>"  
    + " genero :" + req.body.kind + "<br>"  
    + " precio: " + req.body.price);  
});
```



# 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 de 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
```

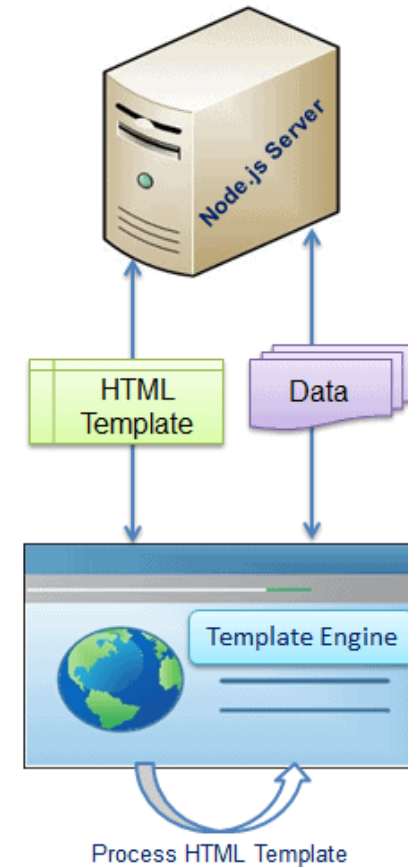
```
let app = express();
// Caso 1.
app.use(express.static('public'));
// acceso.

http://localhost:3000/images/user.png
http://localhost:3000/stylesheets/style.css

// Caso 2.
app.use('static', express.static('public'));
// acceso.
http://localhost:3000/static/images/user.png
http://localhost:3000/static/stylesheets/style.css
```

# Motores de plantillas

- Los motores de plantillas facilitan la **separación de la interfaz de usuario y los datos** (contenido) en una aplicación web.
- Permiten procesar un texto (datos del modelo) y convertirlo en HTML de **forma dinámica y ágil**.
- Tienen acceso a los **atributos del modelo**, pudiendo:
  - Insertarlos en el código HTML o JavaScript.
  - Utilizarlos en estructuras de control (condicionales, bucles, etc.)
- Además, ofrecen sistemas de **composición de plantillas** basado en herencia para fomentar la reutilización de código.



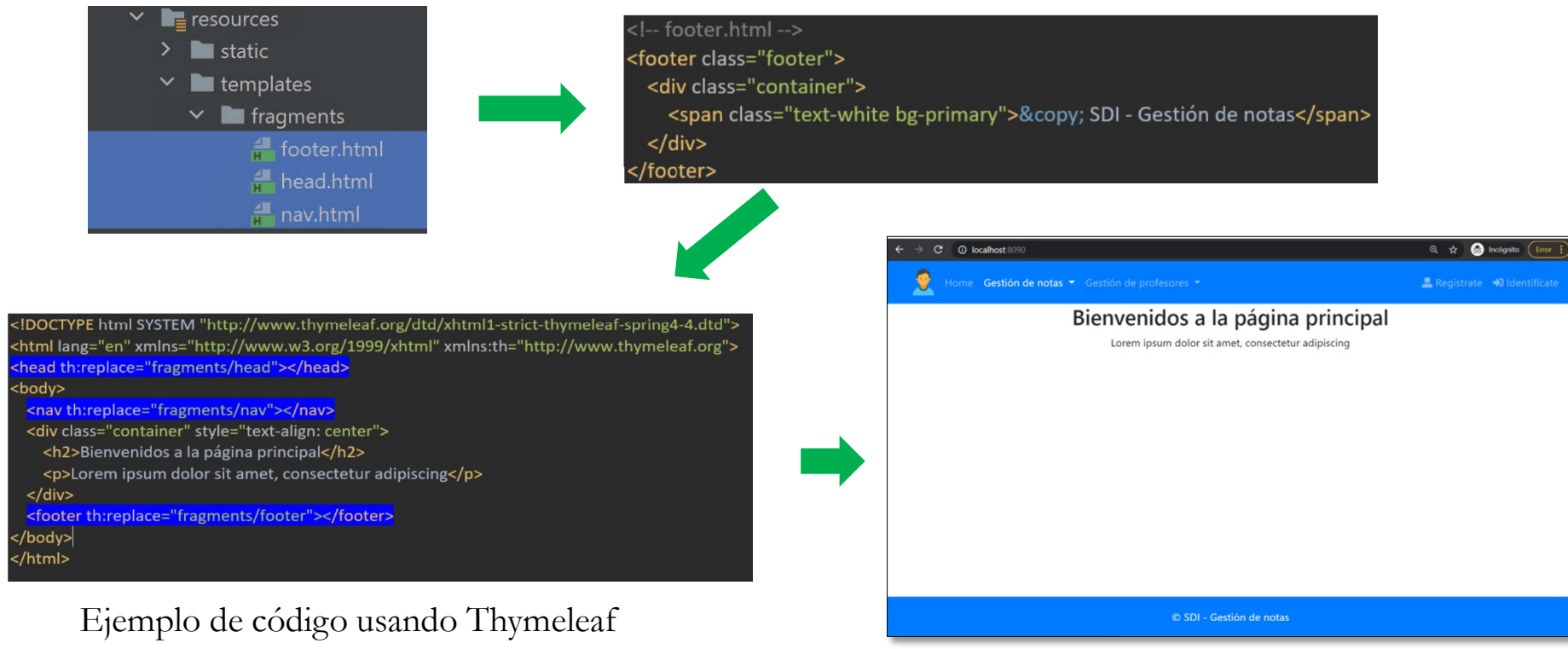
# ¿Por qué usar un motor de plantillas?

- Sintaxis sencilla.
- Además de insertar datos, contienen estructuras de control básicas (if, for, expresiones lógicas, etc.)
- Poseen componentes muy avanzados para incluir en las vistas
  - Sistemas de paginación, fragmentos de AJAX, internacionalización, división de plantillas en bloques, etc.
- Reutilizable, separación total entre la vista y la lógica.
- Ofrecen sistema de herencia y redefinición de bloques, permiten componer plantillas a partir de otras
  - Evitan replicar partes comunes en varias vistas (cabeceras, menús, pie de página, etc.).
- El mismo motor de plantillas puede ser **utilizado en diferentes frameworks**.
- **Mejoran la arquitectura y el mantenimiento.**
- Fomentan la **reutilización de código.**
- **Facilitan las tareas de los desarrolladores .**



# Motores de plantillas

- Sistema de herencia y redefinición, inclusión o sustitución de bloques



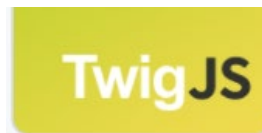
Ejemplo de código usando Thymeleaf



# Express > Vistas y plantillas

- Comúnmente las plantillas se almacenan en el directorio **views**
- La mayor parte de motores de plantillas deben ser instalados, nosotros utilizaremos **Twig**
- Cuando usamos el generador de express ya incluye el motor de plantilla indicado en la generación, si no, podemos instalarlo con npm:

```
npm install twig
```



```
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 (atributos)**

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

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

Plantilla twig

```
{ "seller" : "uniovi" }
```

Atributos del modelo



# Express > Vistas y plantillas

- **Definición de plantillas (objetos)**

- 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>{{ seller.name }}</h1>
  </body>
</html>
```

Plantilla twig

```
{ "seller" :
  {
    "name" : "uniovi",
    "year" : 2000
  }
}
```

Atributos del modelo



# Express > Vistas y plantillas

- **Definición de plantillas (bucles)**

- 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 song in songs %}
      <li>
        {{ song.name }} -
        {{ song.price }}
      </li>
    {% endfor %}
  </ul>
</body>
```

Plantilla twig

```
{ "songs": [
  { "name" : "Blank space",
    "price" : "1.2" },
  { "name" : "See you again",
    "price" : "1.3" },
  { "name" : "Uptown Funk",
    "price" : "1.1" }
]}
```

Atributos del modelo



# Express > Vistas y plantillas

- **Definición de plantillas (estructuras condicionales)**

- Ofrece **estructuras condicionales** para incluir código en base a una expresión lógica, la sintaxis es la siguiente:
  - `{% if <expresión lógica> %} {% endif %}`

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

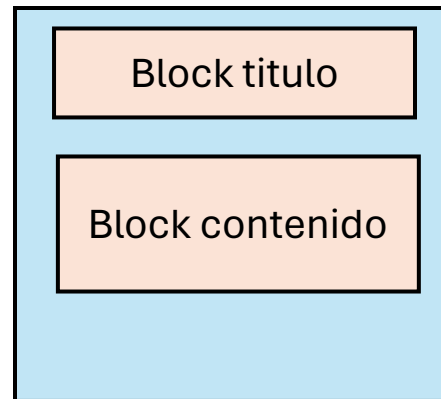
- Soporta casi el mismo conjunto de expresiones lógicas definido en Javascript
- Documentación completa:  
<https://www.npmjs.com/package/twig>



# Express > Vistas y plantillas

- **Definición de plantillas (composición de plantillas)**

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



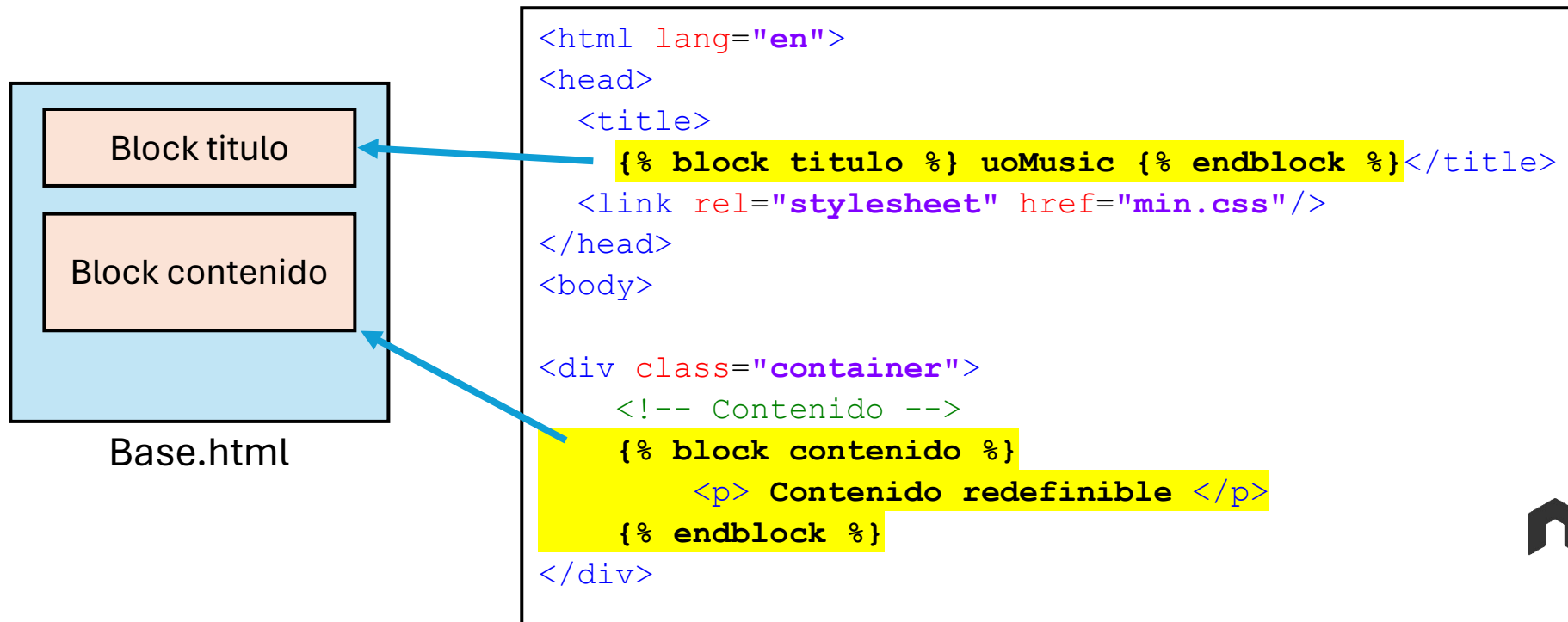
Base.html



# Express > Vistas y plantillas

- **Definición de plantillas (composición de plantillas)**

- Ejemplo, declaración de bloques en plantilla *base.html*



# 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 song in songs %}  
        <li>{{ song.name }} </li>  
    {% endfor %}  
</ul>  
{% endblock %}
```



# Express > Vistas y plantillas

- **Renderización de plantillas en la aplicación**
  - Incluyendo el motor de plantilla Twig en la aplicación
    - Se incluye en **app.js** junto al resto de módulos globales

```
let express = require('express');
let app = express();
...

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'twig');
..
```





# Express > Vistas y plantillas

- **Renderización de plantillas en la aplicación**

- El objeto **twig** contiene una función **render(<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

```
app.get("/songs", function (req, res) {  
  songs = [  
    {'title': 'Blank space', 'price': '1.2'},  
    {'title': 'See you again', 'price': '1.3'},  
    {'title': 'Uptown Funk', 'price': '1.1'}]  
  let response = {  
    seller: 'Tienda de canciones',  
    songs: songs  
  };  
  res.render('test', response);  
});
```

```
<h1> {{ seller }} </h1>  
  
{% for song in songs %}  
  <li>{{ song.name }} </li>  
{% endfor %}
```

test.twig





Escuela de Ingeniería Informática

Escuela de Ingeniería Informática  
School of Computer Science Engineering

Universidad de Oviedo  
*Universidá d'Uviéu*  
*University of Oviedo*

# Sistemas Distribuidos e Internet

## Tema 7 Introducción a Node.js



**Dr. Edward Rolando Núñez Valdez**

nunezedward@uniovi.es