



# Iniciación a NodeJS



## Índice

- 0. Consideraciones previas
- 1. Introducción
- 2. Módulos NodeJS
- 3. Módulo express.js
- 4. Ejemplo: Servidor de imágenes
- 5. Ejemplo: CRUD básico (extra)



## **1. Introducción**

- 1. ¿Qué es NodeJS?**
- 2. Diferencias entre JS y NodeJS**
- 3. Instalación NodeJS**
- 4. Hola mundo**
- 5. Características principales de NodeJS**
- 6. Const, Let y Var**
- 7. Node Package Manager (NPM)**
- 8. Programación Asíncrona con Javascript**



## **2. Módulos NodeJS**

- 1. Global**
- 2. Process**
- 3. Módulo HTTP**
- 4. Eventos NodeJS**
- 5. E/S de archivos con NodeJS**
- 6. Variables de entorno**



### **3. Módulo express.js**

- 1. ¿Qué es?**
- 2. Instalación y configuración**
- 3. Gestión de peticiones**
- 4. Enrutamiento**
- 5. Creación de una API**
- 6. Creación de un sitio web**
- 7. Jade (Motor de plantillas)**



## 4. Ejemplo: Servidor de imágenes



## 5. Ejemplo: CRUD básico



## **(o) Consideraciones previas**

En este capítulo vamos a hablar de varias herramientas que nos van a facilitar la vida en cuanto al desarrollo y depuración de las aplicaciones que realicemos.

Cuanto más usemos estas, mejor será nuestro desarrollo.





## (o) Consideraciones previas

**Visual Studio Code** - IDE de desarrollo. Conjunto de herramientas para el desarrollo de aplicaciones, no solo de NodeJS sino de muchos otros lenguajes de programación. Es gratuito y desarrollado por Microsoft y tiene infinidad de plugins que nos ahorrarán mucho trabajo.

<https://code.visualstudio.com/>



## (o) Consideraciones previas

**Postman** - Gestor de peticiones HTTP para generar, probar y estudiar nuestras APIs (o de terceros) y no necesitas constantemente la ayuda de un navegador. Aunque en este curso lo veremos poco, vendría bien familiarizarse con el mismo.

<https://www.postman.com/downloads/>

(Ej. <https://api.publicapis.org/entries>)



## (o) Consideraciones previas

**Chrome (Dev Tools)** - Google Chrome es un navegador web, que además ofrece las Developers Tools que es un conjunto de herramientas básicas para comprender y depurar los sitios web.

Inspector CSS

Debugger

Watcher

Console



## (o) Consideraciones previas

**GIT** - GIT es un software de control de versiones ampliamente usado y extendido. En el curso no haremos uso de él, pero es una herramienta recomendada tanto si eres un único desarrollador como si formas parte de un equipo de trabajo.

Hay varios clientes gratuitos:

<https://github.com/>

<https://bitbucket.org/>



## (1) ¿Qué es NodeJS?

Node.js no es un lenguaje de programación. Es un entorno de ejecución que se utiliza para ejecutar JavaScript fuera del navegador.

Tanto JavaScript como Node.js se ejecutan en el motor de tiempo de ejecución JavaScript V8 creado por Google.



## (1) Diferencias entre JavaScript y NodeJS

- JS es un lenguaje de programación
- JS solo se ejecuta en un navegador
- Se usa en el lado del cliente
- Podemos interactuar con el DOM y CSSDOM
- Funciona en cualquier navegador con motor de JS.
- NodeJS es un entorno de ejecución de JS
- Con NodeJS podemos ejecutar JS en cualquier sitio.
- Se ejecuta mayoritariamente en servidores
- No hay HTML tags ni nada similar
- Usa JS V8 engine.



## (1) Instalación NodeJS

<https://nodejs.org/en/>

Versión LTS o Versión current

<https://nodejs.org/en/about/releases/>

## (1) Instalación NodeJS







## (1) Instalación NodeJS

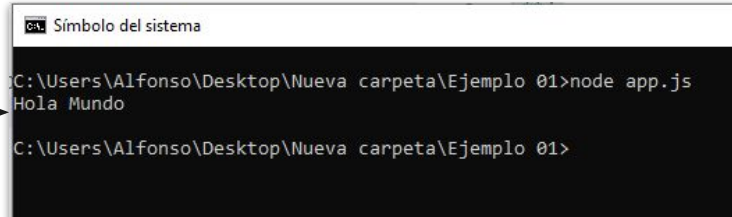
¿Cómo gestionar distintas versiones?

nvm (Node.js Version Manager) [linux]

<https://github.com/coreybutler/nvm-windows> (windows)

## (1) Hola Mundo (/Ejemplo\_01\_01)

```
/*  
 * Primera aplicación NodeJS  
 **/  
let myVar = 'Hola Mundo';  
console.log(myVar);
```



A terminal window titled "Símbolo del sistema" (System Symbol) showing the execution of a Node.js script. The command prompt is "C:\Users\Alfonso\Desktop\Nueva carpeta\Ejemplo 01>node app.js". The output is "Hola Mundo". The prompt is "C:\Users\Alfonso\Desktop\Nueva carpeta\Ejemplo 01>".

```
Símbolo del sistema  
C:\Users\Alfonso\Desktop\Nueva carpeta\Ejemplo 01>node app.js  
Hola Mundo  
C:\Users\Alfonso\Desktop\Nueva carpeta\Ejemplo 01>
```

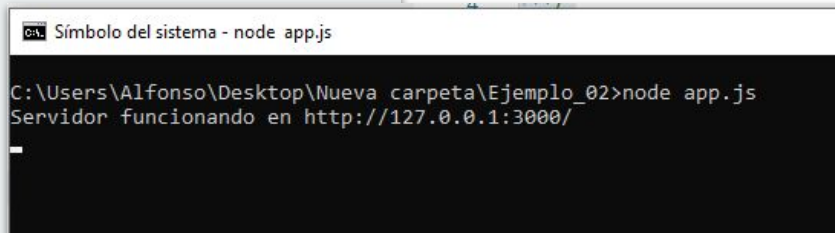
## (1) Hola Mundo (/Ejemplo\_01\_02)

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

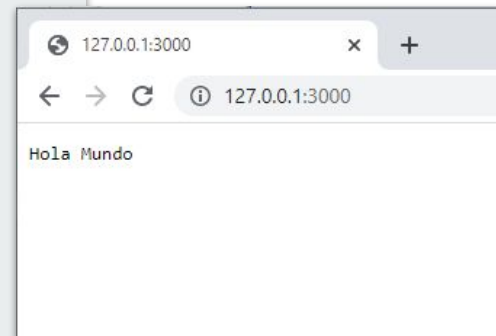
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hola Mundo');
});

server.listen(port, hostname, () => {
  console.log(`Servidor funcionando en http://${hostname}:${port}/`);
});
```



Símbolo del sistema - node app.js

```
C:\Users\Alfonso\Desktop\Nueva carpeta\Ejemplo_02>node app.js
Servidor funcionando en http://127.0.0.1:3000/
```





## (1) Características principales de NodeJS

- Asíncrono y controlado por eventos: todas las API de la biblioteca Node.js son asíncronas, es decir, sin bloqueo.
- Procesos en un solo hilo pero altamente escalable gracias a su sistema de eventos y bucle.
- Puede atender muchas más peticiones que otros servidores tradicionales (Apache, IIS)



## (1) Const, Let y Var (/Ejemplo\_01\_03)

**CONST:** Es una constante la cual **NO** cambiará su valor en ningún momento en el futuro.

```
const myConst = 'ESTO ES UNA CONSTANTE';  
console.log(myConst);  
  
// Aquí lanzará una excepción  
myConst = 'OTRA COSA';  
console.log(myConst);
```



## (1) Const, Let y Var (/Ejemplo\_01\_03)

**VAR:** Es una variable que **SÍ** puede cambiar su valor y su scope es local.

```
var myVar = 'ESTO ES UNA VARIABLE';
console.log(myVar);

if (true) {
  var myVar = 'CAMBIAMOS SU VALOR DENTRO DE ESTE BLOQUE';
  console.log(myVar);
}

console.log(myVar);
```



## (1) Const, Let y Var (/Ejemplo\_01\_03)

**LET:** Es una variable que **SÍ** puede cambiar su valor, pero solo vivirá (funcionara) en el bloque donde fue declarada.

```
let myVar = 'ESTO ES UNA VARIABLE';  
console.log(myVar);  
  
if (true) {  
  let myVar = 'CAMBIAMOS SU VALOR DENTRO DE ESTE BLOQUE';  
  console.log(myVar);  
}  
  
console.log(myVar);
```



## (1) Node Package Manager (NPM) (/Ejemplo\_01\_03)

Es un gestor de paquetes para Node.js

Gracias a él, los desarrolladores pueden crear, compartir y reutilizar módulos en sus aplicaciones.

<https://www.npmjs.com/>

NPM viene por defecto en los instaladores de nodejs, no obstante si has optado por compilar tu misma versión puedes descargar NPM desde su web e instalarlo por separado.





## (1) Node Package Manager (NPM) (/Ejemplo\_01\_03)

Podemos instalar cualquier paquete a través del comando

***npm install [nombre\_módulo]***

Con esto, instalaremos la última versión del módulo, y debemos recibir una salida de este tipo:

***module\_name@version ./node\_modules/nombre\_módulo***



## (1) Node Package Manager (NPM) (/Ejemplo\_01\_03)

Los módulos se pueden instalar de manera local (por defecto) o de manera global.

Al instalarlos de manera local se instalarán en la carpeta del proyecto, mientras que si los instalamos de manera global se instalarán en la carpeta donde esté instalado node, y estos podrán ser utilizados por cualquier aplicación que se ejecute en el sistema.

Para instalar de manera global, debemos añadir la etiqueta -g

***npm install -g [nombre\_módulo]***



## (1) Node Package Manager (NPM) (/Ejemplo\_01\_03)

El archivo **package.json**, generado por el NPM es muy importante y no solo guardará información sobre el proyecto, también almacena información sobre los módulos y dependencias de los mismos para mantener actualizado el proyecto siempre y en cualquier máquina.

De esta manera, para almacenar una aplicación en un repositorio de versiones (GIT) no hace falta almacenar la carpeta `./node_modules` y sus miles de archivos, ya que estos pueden ser instalados en cualquier momento a partir del archivo **package.json** mediante el comando:

***npm install***



## **(1) Programación Asíncrona con Javascript** (/Ejemplo\_01\_04)

Para explicar la programación asíncrona, empezaremos explicando qué es la programación síncrona.

Programación síncrona es aquella que detiene la ejecución de la aplicación en cada tarea hasta que termina.

Si por ejemplo vamos a leer un archivo, y tarda 250ms, la aplicación estará parada 250ms, y una vez ha leído el archivo, continúa con la lógica.



## (1) Programación Asíncrona con Javascript (/Ejemplo\_01\_04)

Si trasladamos el ejemplo anterior a programación asíncrona equivale a decir que el programa comienza a leer el archivo y cuando acabe, ejecutará una función de *callback* o *promesa*, mientras tanto continúa con la ejecución de la aplicación.



## (1) Programación Asíncrona con Javascript (/Ejemplo\_01\_05)

Javascript es un lenguaje que utiliza la **programación asíncrona dirigida por eventos** para el tratamiento de la concurrencia, es decir para realizar más de una tarea simultáneamente. Es común escuchar que javascript se ejecuta en un sólo hilo, pero esto es una verdad a medias.

NodeJS utiliza el bucle de eventos para trabajar con la programación asíncrona. Básicamente permite almacenar las funciones de callback y ejecutarlas cuando corresponde, es decir cuando tenga lugar el evento que corresponda



## (2) Global (/Ejemplo\_02\_01)

Del mismo modo que en un navegador nos encontramos el objeto **window**, en Nodejs tenemos el objeto **global**.

La variable global es accesible por cualquier script o programa de node y tiene distintas propiedades como *global.process*, *global.require* o *global.console*

Cualquier propiedad de primer nivel de global es accesible sin tener que referirse a ella con el prefijo global.



## (2) Global (/Ejemplo\_02\_01)

Algunos de las propiedades/funciones principales de global son:

- process
- require
- module
- console
- setTimeout/setInterval
- **\_\_dirname** - Obtenemos la ruta del directorio donde se está ejecutando la aplicación
- **\_\_filename** - Obtenemos la ruta junto con el nombre del archivo donde se está ejecutando la aplicación





## (2) Process (/Ejemplo\_02\_02/app.js)

Cuando ejecutamos algo en un ordenador, estamos usando un proceso. Como ya hemos visto, NodeJS se ejecuta en un único proceso. El sistema operativo asigna un identificador único a los procesos, llamado pid.

Con el módulo process, NodeJS dejará al desarrollador interactuar con este proceso y realizar algunas tareas importantes para el correcto funcionamiento de las aplicaciones.



## (2) Process

Dos de las características más utilizadas del módulo *process* son:

- Para pasar argumentos a la aplicación
- Para realizar tareas cuando se ha producido un error o bien cuando la aplicación termina su ejecución.



## (2) Process (/Ejemplo\_02\_02/app\_argv.js)

Algunas aplicaciones necesitan argumentos para su funcionamiento, ya sea un archivo txt o un usuario|contraseña para conectar con servicios de terceros etc...

Esto lo podemos hacer ejecutando la aplicación con el siguiente comando

***node app.js argv1 argv2 argv3***

En la aplicación, podemos obtener los argumentos mediante *process.argv* y actuar en consecuencia.



## (2) Process (/Ejemplo\_02\_02/app\_on.js)

El módulo process, como hemos comentado nos proporciona dos eventos que pueden ser de gran utilidad.

`process.on('exit', function() {})`; Se lanza siempre que la aplicación termina su ejecución.

`process.on('uncaughtException', function(err) {})`; Se lanza siempre que la aplicación sufre una excepción inesperada.



## (2) Módulo HTTP (/Ejemplo\_02\_03)

NodeJS tiene un módulo nativo destinado para la creación y explotación de servidores HTTP de bajo nivel para programación de aplicaciones API (por ejemplo).

Es sumamente fácil crear un servidor HTTP con NodeJS, de modo que los desarrolladores solo deben preocuparse de la lógica de la propia aplicación (API) sin preocuparse de otros aspectos relativos a la arquitectura misma del servidor.



## (2) Módulo HTTP (Ejemplo\_02\_03)

El ejemplo más simple de servidor web lo podemos ver a continuación:

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hola Mundo');
});

server.listen(port, hostname, () => {
  console.log(`Servidor funcionando en
http://${hostname}:${port}/`);
});
```



## (2) Módulo HTTP (/Ejemplo\_02\_04)

Con NodeJS y el módulo HTTP también podemos crear un servidor de páginas web, como puede ser Apache, IIS, Nginx... no obstante aunque no es recomendable, veremos a continuación un ejemplo.

En el Ejemplo\_02\_04 podemos ver un servidor web que responderá con el contenido de un archivo .html si hacemos una petición a la raíz (<http://127.0.0.1:3000/>) y con un error 404 a cualquier otra petición.



## (2) Módulo HTTP (/Ejemplo\_02\_05)

También, y para poder mostrar todo lo que podemos hacer, vamos a añadir complejidad al servidor HTML. Vamos a añadir un formulario, donde el usuario podrá introducir su *nombre* y *email* y lo procesaremos en el servidor.

En el Ejemplo\_02\_05 podemos ver un servidor web que responderá con el contenido de un archivo .html, donde encontraremos un formulario simple para que el usuario envíe sus datos.

Una vez enviados, interceptamos la petición y mostraremos por pantalla la información introducida por el usuario.





## (2) Módulo HTTP

Como hemos visto, NodeJS es capaz de crear un servidor http de bajo nivel, y depende de los desarrolladores dotar con mayor o menor funcionalidad al mismo.

Realizar un servidor web HTML quizá no tenga mucho sentido, pero si servidores API o muy específicos para realizar determinadas tareas.

Para realizar servidores web, más adelante aprenderemos el uso de algunos frameworks que facilitan enormemente la labor, como es ***express.js***



## (2) Módulo HTTP (/Ejemplo\_02\_06)

El módulo HTTP, no solo sirve para gestionar peticiones entrantes, también nos sirve para realizar peticiones a servidores de terceros para obtener información.

Imaginemos una aplicación encargada de chequear cada X minutos si un servidor está OK o KO, en caso de estar KO enviar una alerta a quien corresponda.

Esto lo podemos realizar con el módulo HTTP como podemos ver en el Ejemplo\_02\_07



## (2) Eventos NodeJS (/Ejemplo\_02\_07)

NodeJS es un sistema de eventos, gracias a los cuales es capaz de soportar gran cantidad de procesos simultáneos sin sufrir bloqueos.

Con el uso de los distintos módulos, veremos que llevan asociados eventos, los cuales podemos usar para dar funcionalidad a nuestras aplicaciones.

Vamos a ver un ejemplo simple, con el que gracias a los eventos nos sufriremos bloqueos y podremos resolver grandes problemas.



## (2) Eventos NodeJS (/Ejemplo\_02\_07)

Vamos a imaginar que tenemos un archivo de entrada con datos, y tenemos que procesar los datos y sacar algunas conclusiones. Lo normal sería cargar el archivo en memoria e ir procesando, pero podemos tener un problema de rendimiento si el archivo es muy grande.

La opción pasa por ir leyendo línea a línea, y por cada línea leída podemos realizar los cálculos específicos de la misma, así descargamos el servidor y podremos continuar con muchos trabajos en paralelo.



## (2) E/S de archivos con NodeJS (/Ejemplo\_02\_08)

La lectura/escritura de archivos es una constante en todos los programas en cualquier lenguaje. Esta tarea en NodeJS será asíncrona (como de costumbre) para soportar la concurrencia.

Ejemplos de lectura de archivos hay muchísimos, como el ejemplo anterior, donde teníamos que leer un archivo de entrada para procesar sus datos.

Para la escritura de datos, podemos pensar en el típico log de la aplicación, donde cada proceso puede ir escribiendo información, errores y simples mensajes para los desarrolladores durante su ejecución, con lo que debemos contemplar que varios puntos de la aplicación puedan escribir de manera simultánea.



## (2) Variables de entorno (/Ejemplo\_02\_09)

Las variables de entorno son variables externas a nuestra aplicación que residen en el sistema operativo o en el contenedor de la aplicación que se está ejecutando. Una variable de entorno es simplemente un nombre asignado a un valor.

Por convención, el nombre se escribe con mayúscula y los valores son cadenas de texto, por ejemplo: `PORT=8080`.

Normalmente estas variables son almacenadas en un archivo llamado `.env`, teniendo uno para desarrollo y otro para producción.



## (2) Variables de entorno

Ej. desarrollo

```
NODE_ENV=development  
HOST=127.0.0.1  
PORT=8080
```

Ej. producción

```
NODE_ENV=production  
HOST=127.0.0.1  
PORT=80
```

Estos archivos se deben gestionar para cargarlos en la aplicación, o bien manualmente o bien a través de algún módulo, como **dotenv**

***npm i dotenv***



## (2) Variables de entorno

Para gestionarlo, simplemente tenemos que cargar el módulo, o bien generar uno propio (config.js) donde podremos establecer los valores por defecto, en caso de no existir en el archivo **.env**

```
const dotenv = require('dotenv').config();

module.exports = {
  NODE_ENV: process.env.NODE_ENV || 'development',
  HOST: process.env.HOST || '127.0.0.1',
  PORT: process.env.PORT || 8080
}
```





### (3) ¿Qué es express.js?

Express es un framework Web para Node.js. Las aplicaciones Web comparten patrones, por eso es conveniente utilizar un framework. Gracias a él, desarrollaremos aplicaciones más estables en menos tiempo.

Es un framework muy ligero que nos dará las herramientas básicas para crear una aplicación muy potente.



### (3) ¿Qué es express.js?

Con Express podemos, con poco esfuerzo, realizar:

- API basadas en JSON
- Aplicaciones Web de una sola página
- Aplicaciones Web que se ejecutan en tiempo real.



### (3) ¿Qué es express.js?

Las principales ventajas son:

- Reduce el tiempo necesario para crear aplicaciones.
- Contiene modelos, como enrutamientos y capas para vistas, con lo que no es necesario volver a programarlos.
- Tiene una comunidad de desarrolladores grandísima, con lo que su código es bastante estable y está actualizándose constantemente.



### (3) Instalación y configuración

Debemos instalar express.js desde NPM mediante el comando

***npm install express***

***npm install express-generator -g***

Nótese el uso de la etiqueta -g para instalar el generador de manera global. Con este generador podremos generar un sitio Web a partir de una plantilla para comenzar el desarrollo.



## (3) Gestión de peticiones

Como todo servidor HTTP, debemos ser capaces de gestionar las distintas peticiones que reciba nuestro servidor.

Estas peticiones pueden ser de distintos tipos, contener o no parámetros, y esto `express.js` lo soluciona mediante el enrutamiento, que veremos a continuación.



## (3) Enrutamiento

El enrutamiento describe cómo responderá una aplicación a las solicitudes HTTP. Cada vez que interactuamos con una aplicación o sitio Web, el navegador se encarga de realizar estas solicitudes.

Express usa comandos HTTP para describir las rutas, los más comunes son **GET** y **POST**, pero puede trabajar con otros como DELETE, PUT, HEAD, OPTIONS y TRACE



## (3) Enrutamiento

Si queremos que nuestra aplicación responda de una determinada manera cuando un cliente solicita la url /about, tendremos que crear una ruta para tratar estas peticiones, y lo haremos de la siguiente manera.

```
[...]  
app.get('/about', (req, res) => {  
  res.send('Estamos en /about');  
});
```



## (3) Enrutamiento

Si queremos que nuestra aplicación procese datos recibidos mediante POST en la URL `/create` para almacenar cierta información de manera persistente, debemos configurar la ruta de la siguiente manera

```
[...]  
app.post('/create', (req, res) => {  
  res.send(req.body);  
});
```





## (3) Enrutamiento

Muchas veces estamos viendo URLs 'dinámicas', es decir que cambian su contenido en función de algún parámetro de la URL. En express también podemos tratar estas URLs de una manera fácil y amigable para devolver al usuario la información deseada.

```
[...]
app.get('/get/:id', (req, res) => {
  res.send('Devolvemos el id: ' + req.params.id);
});
```



### **(3) Creación de una API** (/Ejemplo\_03\_01)

Como hemos comentado anteriormente, NodeJS está pensado para aplicaciones de alta concurrencia como puede ser una API de consulta sobre determinados temas. En el siguiente ejemplo podemos ver una API muy simple pero suficiente para hacernos una idea.

Tenemos una lista de ciudades por países con algunos datos, que podremos consultar mediante algunas URLs y ver los resultados.



### (3) Creación de una API (/Ejemplo\_03\_01)

Las URLs disponibles son:

<http://localhost/> - Devuelve el status del servidor API

<http://localhost/getByCountry/{Pais}> - Devuelve las ciudades de un país

<http://localhost/getByName/{Name}> - Devuelve las ciudades que contienen una determinada cadena en su nombre.

Todas las peticiones son GET, pero podríamos soportar GET, POST, DELETE, PUT... sin problemas estableciendo las rutas correspondientes.



## (3) Creación de una API - Middlewares (/Ejemplo\_03\_01\_auth)

Las funciones de middleware son funciones que tienen acceso al objeto de solicitud (req), al objeto de respuesta (res) y a la siguiente función de middleware en el ciclo de solicitud/respuestas de la aplicación. La siguiente función de middleware se denota normalmente con una variable denominada next.

Las funciones de middleware pueden realizar las siguientes tareas:

- Ejecutar cualquier código.
- Realizar cambios en la solicitud y los objetos de respuesta.
- Finalizar el ciclo de solicitud/respuestas.
- Invocar el siguiente middleware en la pila.



### (3) Creación de una API - Middlewares (/Ejemplo\_03\_01\_auth)

Si la función de middleware actual no finaliza el ciclo de solicitud/respuestas, debe invocar `next()` para pasar el control a la siguiente función de middleware. De lo contrario, la solicitud quedará colgada.

Una posible mejora para la API sería dotarla de un sistema de autorización para que solo usuarios autenticados pudieran usar la misma.



### (3) Creación de una API - Middlewares (/Ejemplo\_03\_01\_auth)

Esto se puede realizar mediante JWT, generando un token que adjuntamos a las peticiones y en el lado del servidor validamos.

<https://jwt.io/>

[https://es.wikipedia.org/wiki/JSON\\_Web\\_Token](https://es.wikipedia.org/wiki/JSON_Web_Token)

Para esto usaremos un middleware que adjuntamos a aquellas peticiones que requieren estar autenticado.



### (3) Creación de un sitio web (/Ejemplo\_03\_02)

Una vez instalado express.js podemos generar la estructura de un sitio Web mediante el siguiente comando:

***express express\_name***

Esto nos guiará en la generación de una nueva aplicación y creará toda la estructura básica para crear un sitio Web, generando la siguiente estructura:



### (3) Creación de un sitio web (/Ejemplo\_03\_02)

**app.js:** Archivo para iniciar la aplicación, contiene información sobre la configuración de la misma.

**node\_modules:** Carpeta donde se instalarán los módulos auxiliares de nuestra aplicación

**package.json:** Archivo donde estará la información de la aplicación y módulos necesarios para el funcionamiento de la misma.





### (3) Creación de un sitio web (/Ejemplo\_03\_02)

**public:** Carpeta que sirve la aplicación web. Aquí encontraremos los archivos “estáticos” de la web.

**routes:** Aquí tendremos la declaración de las rutas que tendrá nuestra aplicación

**views:** Carpeta donde se almacenarán las vistas de la aplicación.

*Esta estructura es una sugerencia, se puede modificar en función de las necesidades de cada proyecto.*



### (3) Jade (Motor de plantillas) (/Ejemplo\_03\_02)

**Jade**, es el motor de plantillas usado por express para compilar y generar el HTML ofrecido por la aplicación.

Est un motor basado en sangrías, con lo que todo su código debe estar correctamente sangrado.

A continuación mostramos un ejemplo comparado con HTML para ver su diferencia y entender mejor su funcionamiento.



### (3) Jade (Motor de plantillas) (/Ejemplo\_03\_02)

```
<div class="wrapper">
  <h1>Ejemplo</h1>
  <p>Vamos a utilizar JADE</p>
  
</div>
```

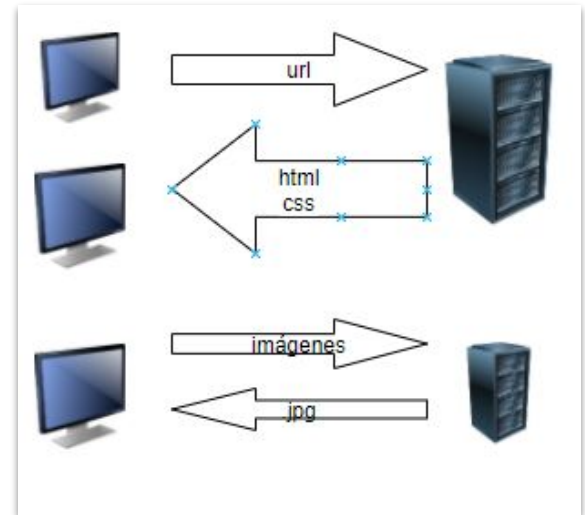
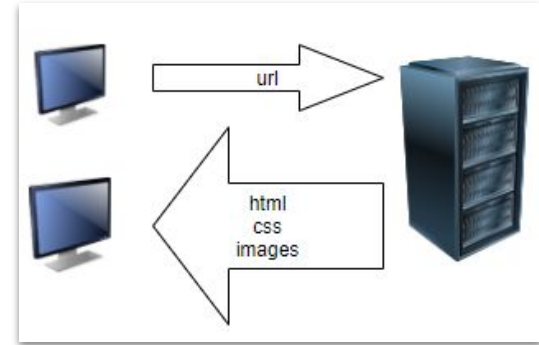
```
.wrapper
  h1 Ejemplo
  p Vamos a utilizar JADE
  img(src='images/image.jpg', alt='Descripción')
```

Podemos encontrar toda la documentación y posibilidades en  
<https://jade-lang.com/api>

## (4) Servidor de imágenes

Imaginemos una aplicación web (tipo Idealista) donde hay miles de peticiones al servidor con búsquedas y solicitudes.

Un mismo servidor tiene que procesar todas las peticiones y en muchas ocasiones pueden ser demasiadas, además de la propia limitación de **nº peticiones por sesión simultáneas de cada servidor web**.





## (4) Servidor de imágenes

Vamos a dividir el proyecto en 3 fases para ir analizando el desarrollo, y viendo como cada una de las fases da funcionalidad al servidor.

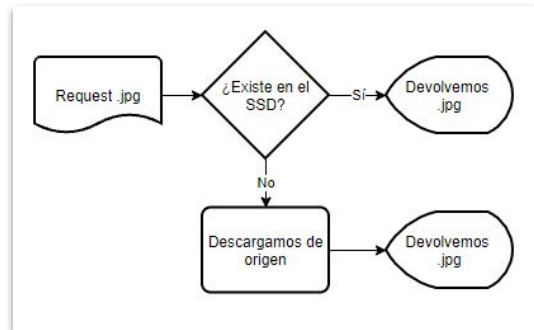
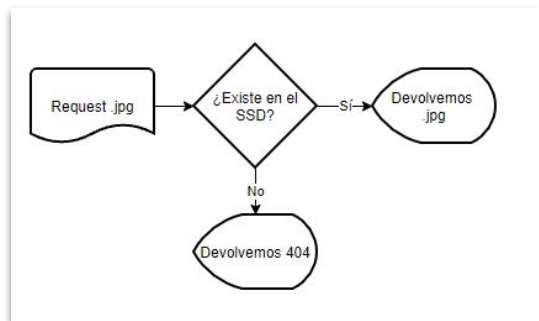
**Fase 1:** Devolvemos la imagen que nos solicitan.

**Fase 2:** Devolvemos la imagen redimensionada a las dimensiones especificadas.

**Fase 3:** Insertamos un sistema de caché para mejorar considerablemente el rendimiento de la CPU.

## (4) Servidor de imágenes (Fase 1) (/Ejemplo\_04 - app.js)

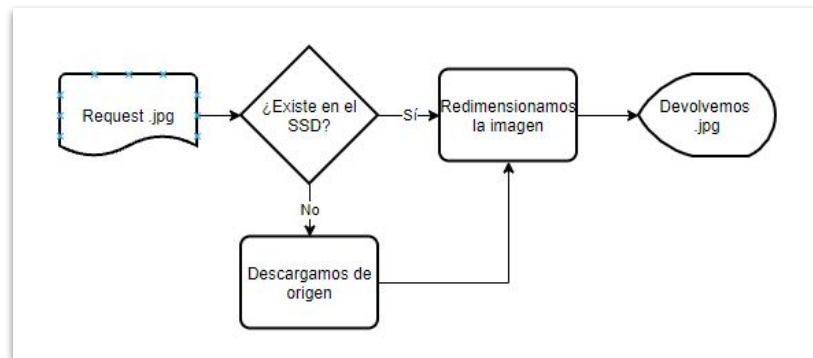
Es la fase inicial y más simple, recibimos una petición, comprobamos si tenemos el archivo de imagen y lo devolvemos.



Posible mejora

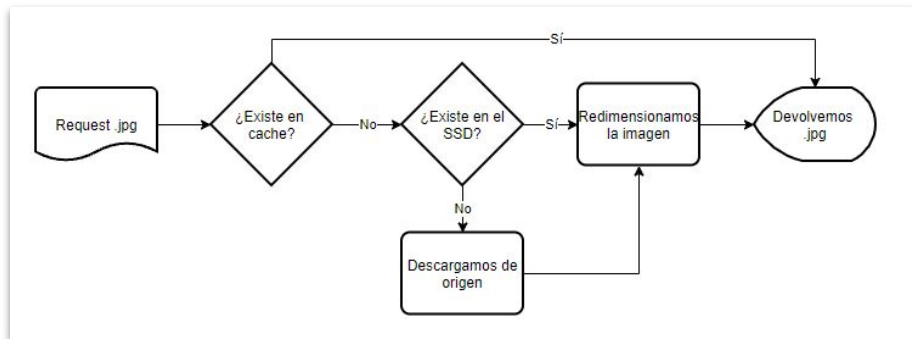
## (4) Servidor de imágenes (Fase 2) (/Ejemplo\_04 - app\_resize.js)

En esta fase, dotamos al servidor de la habilidad para redimensionar en tiempo real las imágenes, de este modo ofreceremos al cliente la imagen deseada sin generar sobrecarga (google lo agradecerá ~ SEO)



## (4) Servidor de imágenes (Fase 3) (/Ejemplo\_04 - app\_cache.js)

En esta fase implementamos un sistema de cache, donde el servidor una vez generadas las imágenes las guarda por si algún otro usuario la solicita más adelante.







## (4) Servidor de imágenes (Mejoras)

Como hemos visto este servidor ha ido evolucionando a lo largo del tiempo, pero aún se pueden hacer muchas mejoras como:

- Devolver formatos .webp.
- Sistema de log/stats
- Borrar la caché no usada cada N días (o con las stats).
- Incluir marca de agua en las imágenes generadas.
- ...



## (5) CRUD básico

CRUD es el acrónimo de “Crear, Leer, Actualizar y Borrar” (Create, Read, Update, Delete)

Normalmente estos sistemas se implementan para tener datos persistentes en el tiempo para su consulta posterior o gestión de la información como:

- Tiendas
- Bibliotecas
- Sistemas de usuarios
- ...



## (5) CRUD básico

Basándonos en express.js, vamos a crear un CRUD que, además, almacenará los datos en una base de datos local en SQLite (por facilidad)

En el mercado hay muchas bases de datos para guardar de manera persistente, tanto SQL como NoSQL.

SQL: MySQL, PostgreSQL, SQLServer, SQLite...

NoSQL: MongoDB, RavenDB, Redis...



## (5) CRUD básico

Para llevar a cabo el proyecto, tendremos que montar un servidor HTML con express, con varias vistas, una para lectura, otra de creación/edición y otra para eliminación.



## (5) CRUD básico (Mejoras)

- Parametrizar todas las consultas SQL (SQL Injection)
- Tratar las excepciones (throw)
- Dotar de estilos css a la aplicación.
- Dotar de sistema de validación de datos (lado Cliente)
- Dotar de sistema de validación de datos (lado Servidor)
- Preguntar antes de eliminar un registro.
- Comprobar emails únicos