Introducción a Node.js

¿Qué es Node.js?

Node.js es un entorno de ejecución del lenguaje *JavaScript* para servidores. Hasta la aparición de *Node.js*, *JavaScript* era un lenguaje que solo lo utilizaban los navegadores web para manipular y controlar elementos HTML, modificar estilos, responder a eventos del usuario y comunicarse con servidores para actualizar dinámicamente el contenido. Uno de los motivos por los que *Node.js* se basó en el uso de *JavaScript*, fue que los desarrolladores pudieran utilizar el mismo lenguaje tanto en el *frontend* como en el *backend* de sus aplicaciones web.

Node.js surgió de la mano de Ryan Dahl en 2009 como una solución para construir aplicaciones web en tiempo real con alta escalabilidad. Dahl lo desarrolló con el objetivo de superar las limitaciones de los servidores tradicionales que estaban diseñados para manejar múltiples solicitudes bloqueantes. *Node.js* adopta un modelo de *Entrada/Salida* no bloqueante y orientado a eventos, lo que permite manejar muchas conexiones simultáneas de forma eficiente. Por ello, resulta ideal para aplicaciones en tiempo real como *chats*, juegos en línea o transmisiones en vivo.

Desde su lanzamiento inicial, *Node.js* ha experimentado un crecimiento significativo y se ha convertido en una herramienta clave en el desarrollo de aplicaciones web modernas.

Características y ventajas

Algunas de las características de *Node.js son*:

- **API basada en eventos**: El enfoque de *Node.js* se basa en eventos, lo que significa que muchas de sus operaciones son manejadas mediante la emisión y escucha de eventos.
- ***Single thread*:** *Node.js* opera en un solo hilo de ejecución principal para realizar todas las operaciones.
- **Eficiente en operaciones E/S**: Su modelo asincrónico y no bloqueante permite manejar múltiples operaciones de entrada y salida de manera eficiente.
- **Escalabilidad**: Altamente escalable ya que permite manejar grandes cantidades de conexiones simultáneas con poco consumo de recursos.
- **Ecosistema robusto**: Cuenta con un amplio ecosistema de librerías y herramientas gracias a *npm* (*Node Package Manager*), lo que facilita la integración de módulos, la reutilización de código y el desarrollo rápido de aplicaciones.
- **Desarrollo rápido**: Al usar *JavaScript* tanto en el *frontend* como en el *backend*, facilita el desarrollo y la colaboración entre equipos.
- **Comunidad activa:** Cuenta con una gran comunidad de desarrolladores que contribuyen con módulos, herramientas y soporte.
- **Adopción de tecnologías modernas**: Compatible con tecnologías modernas como *GraphQL*, *WebSockets*, y *frameworks* como *Express.js*, lo que permite la creación de aplicaciones web avanzadas y escalables.

Arquitectura

En la siguiente figura se representan los principales componentes de la arquitectura de *Node.js* y cómo se relacionan:

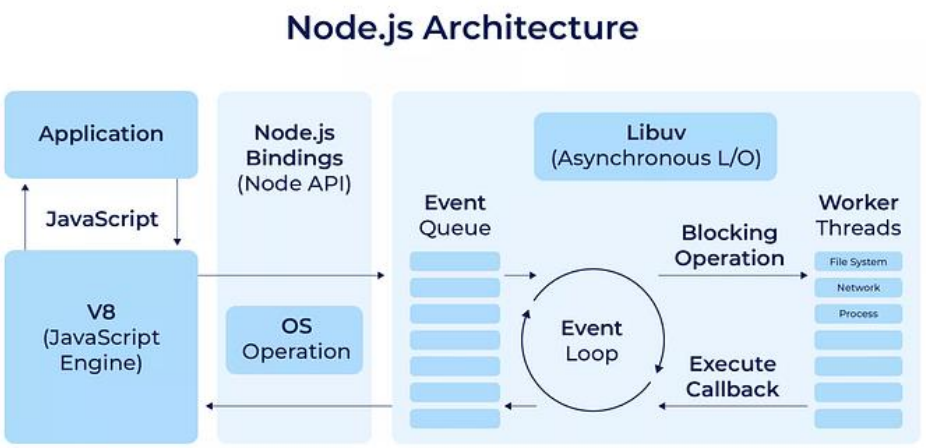


Figura 1: Arquitectura Node.js. Fuente: https://medium.com/@diego.coder/introducci%C3%B3n-a-node-js-b5f33bbe6fb0

- **Motor *V8* de *Google Chrome***: Se encarga de interpretar y ejecutar el código *JavaScript*.
- ***Node.js Bindings***: Interfaz que permite conectar código escrito en *JavaScript* con código escrito en lenguajes nativos como C/C++. Esta API facilita la integración de módulos escritos en lenguajes de bajo nivel con el entorno de ejecución de *Node.js*.
- **Libuv**: Librería multiplataforma con soporte para la programación asíncrona basado en eventos E/S.
- ***Event Loop***: Componente que funciona como un bucle encargado de manejar las operaciones de forma asíncrona.
- **Worker threads:** Los *worker threads* son un conjunto de hilos que permiten ejecutar las tareas más costosas, como lo son el acceso de archivos, criptografía, compresiones o tareas de red, sin bloquear el hilo principal.

Veamos ahora como funciona esta arquitectura:

1. Cada vez que llega una solicitud, *Node.js* la coloca en una cola (*Event Queue*).
2. El *evento loop* espera las peticiones indefinidamente. Cuando llega una solicitud, el bucle la recoge del *event queue* y comprueba si requiere una operación de entrada/salida (E/S) bloqueante. Si no es así, procesa la solicitud y envía una respuesta.
3. Si la solicitud tiene una operación de bloqueo que realizar, el *evento loop* le asigna un hilo disponible del *worker thread*.

4. Cuando la operación ha finalizado *libuv* recibe un evento. Cuando el *event loop* procesa este evento, le indica a *v8* que debe ejecutar el *callback* que tenga asociado.

Cuándo utilizar Node.js

Es interesante utilizar *Node.js* en:

- **Aplicaciones en tiempo real:** Para construir aplicaciones que requieren comunicación en tiempo real, como chats, juegos en línea, aplicaciones colaborativas o sistemas de notificaciones.
- **Aplicaciones basadas en eventos:** Donde la arquitectura basada en eventos y el manejo eficiente de múltiples conexiones simultáneas son fundamentales, como en aplicaciones *IoT* (Internet de las cosas) o sistemas de monitorización en tiempo real.
- **Aplicaciones de streaming:** Para proyectos que manipulan grandes cantidades de datos en tiempo real, como procesamiento de video, audio o transmisiones de datos.
- **APIs y servicios web:** Para construir *APIs RESTful* y servicios web, especialmente cuando se trata de escalabilidad y manipulación de solicitudes concurrentes.
- **Desarrollo rápido de prototipos**: Es ideal para prototipar rápidamente aplicaciones web debido a la facilidad de uso de JavaScript en el *frontend* y el *backend*, así como a su ecosistema de librerías y herramientas.

No es la mejor opción para:

- **Cálculos intensivos en CPU:** Para tareas que requieren mucho procesamiento intensivo en CPU, puede no ser la mejor opción debido a su naturaleza de un solo hilo y su ejecución en un solo núcleo.
- **Aplicaciones heredadas o dependientes de librerías específicas:** Si una aplicación necesita interactuar intensamente con librerías o sistemas que no son compatibles con *Node.js*, podría ser complicado adaptarla.
- **Experiencia del equipo de desarrolladores:** Si tu equipo tiene experiencia significativa en otros lenguajes o tecnologías que son más adecuadas para el proyecto, puede ser más conveniente utilizar esas herramientas en lugar de adoptar Node.js.

Instalación del entorno de trabajo

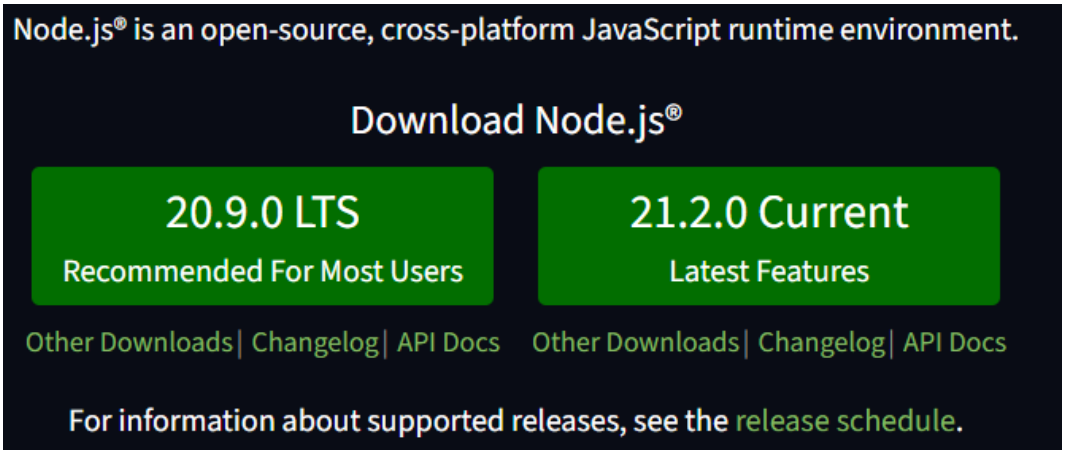
En este apartado prepararemos el entorno de trabajo que utilizaremos en esta unidad. A continuación, se indican los pasos a seguir para realizar la instalación de *Node.js* y el entorno de desarrollo *Visual Studio Code*.

Node.js

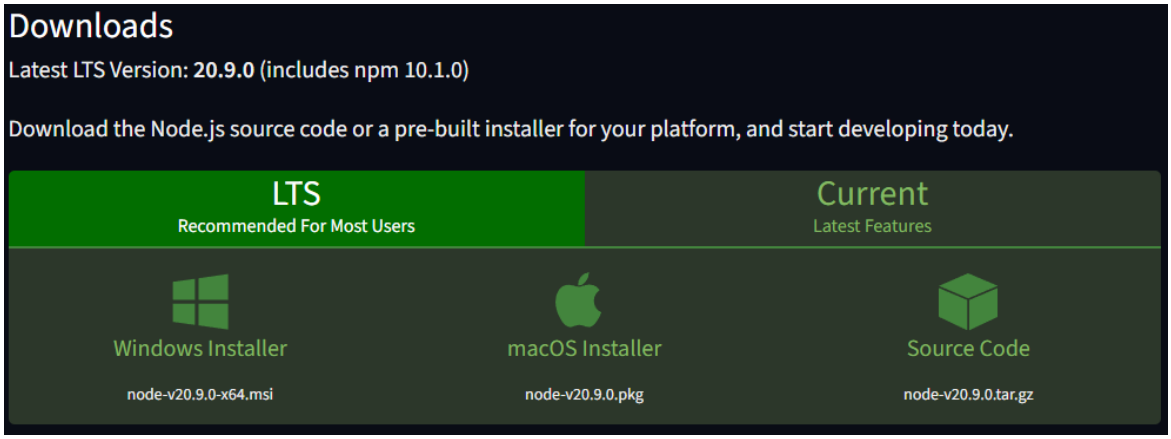
Instalación

Instalar *Node.js* es muy sencillo. En primer lugar, descararemos el instalador desde la página web oficial (http://nodejs.org).

Veremos que hay dos versiones: LTS y Current. LTS (Long Term Support o Soporte a Largo Plazo) es una versión estable y confiable que tiene soporte por un largo período de tiempo. La versión *Current* incluye las últimas características, aunque puede que no sea tan estable. En nuestro caso instalaremos la versión LTS.



En el caso de que estemos utilizando Windows se descargará un instalador con extensión *.msi*. Si nos encontramos en Mac OS se descargará un archivo *.pkg*. Si visitamos la página https://nodejs.org/en/download tendremos la posibilidad de elegir nosotros mismos qué instalador queremos descargar o incluso descargar el código fuente.



Ya sea en Windows o Mac OS únicamente tendremos que ejecutar el instalador y seguir los pasos indicados hasta su finalización. En ningún caso modificaremos los valores por defecto que muestran los diferentes pasos.

Una vez haya finalizado la instalación abriremos la línea de comandos y escribiremos el comando *node -v*. Si todo ha ido bien, mostrará por pantalla la versión que tenemos instalada de *Node.js*.

```
C:\Users\Jesús>node -v
v20.9.0
```

Ejecutando nuestro primer programa

A continuación, escribiremos nuestro primer programa, por supuesto, un hola mundo. Para ello abriremos un bloc de notas o el editor de textos que prefiramos y escribiremos la siguiente línea de código:

```
console.log("Hola mundo!")
```

Guardaremos el archivo con el nombre "*HolaMundo.js*" y desde la línea de comandos iremos al directorio donde se encuentra el archivo. Finalmente ejecutaremos el siguiente comando:

```
node HolaMundo.js
```

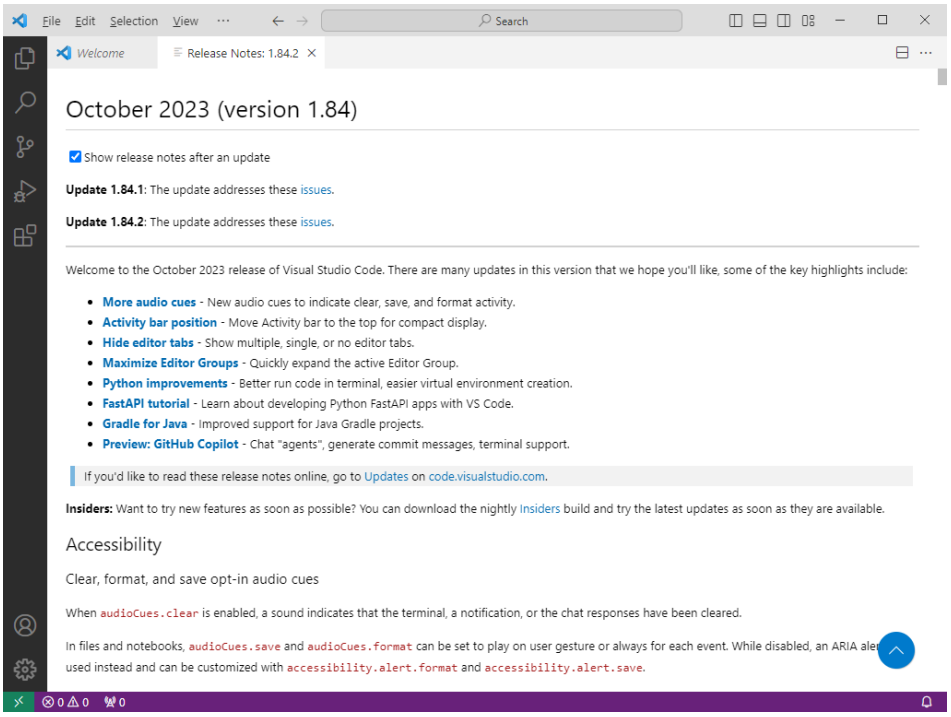
Si todo va bien, se mostrará por pantalla el texto "Hola mundo!".

Visual Studio Code

Visual Studio Code será el entorno de desarrollo (*IDE*) que utilizaremos para desarrollar nuestros proyectos. Existe una gran variedad de entornos que podríamos utilizar. Nosotros utilizaremos Visual Studio Code porque su instalación, configuración y uso es muy sencillo.

Instalación

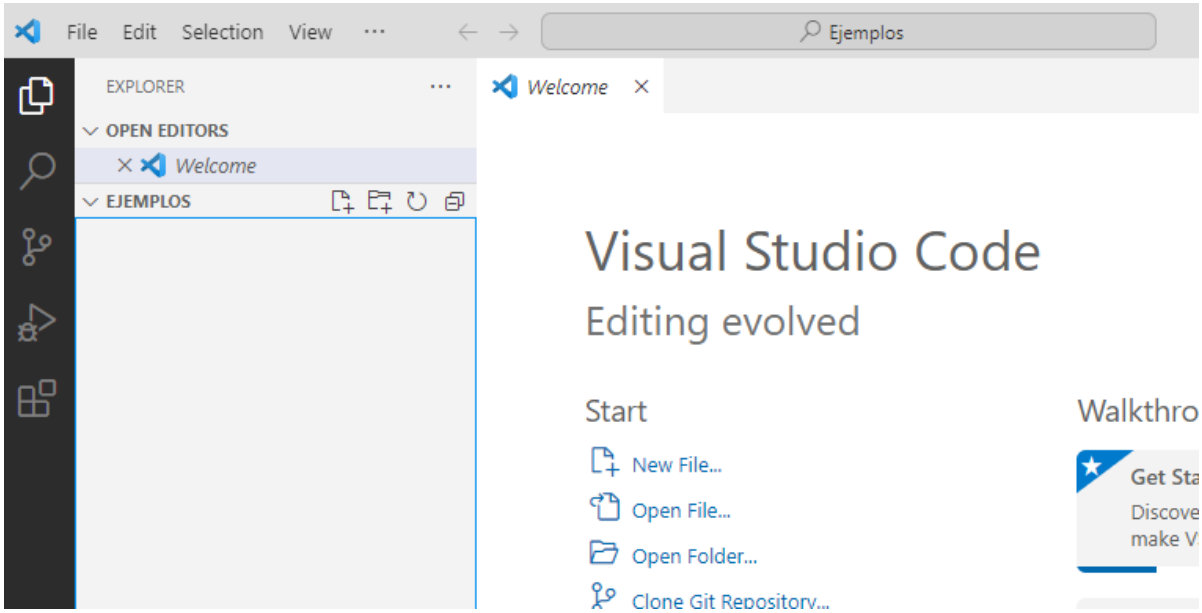
Para instalar Visual Studio Code únicamente tendremos que descargar y ejecutar el instalador desde la página web del proyecto (https://code.visualstudio.com/). Elegiremos el sistema operativo que queramos y seguiremos los pasos que nos indique el instalador. Al abrir el programa se mostrará una pestaña de bienvenida y otra con los cambios que incluye la última versión que tenemos instalada.



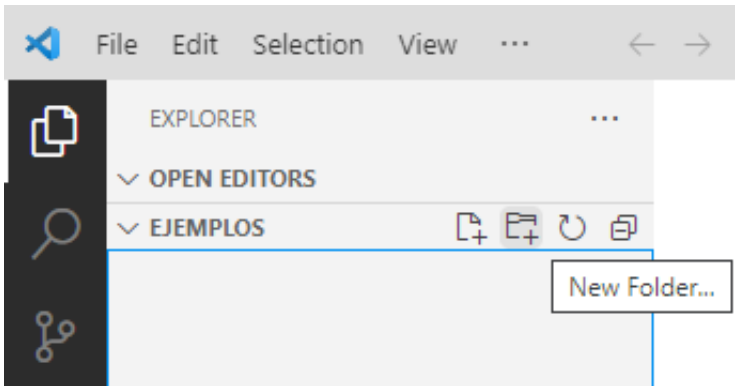
Ejemplo 1 | Primer proyecto

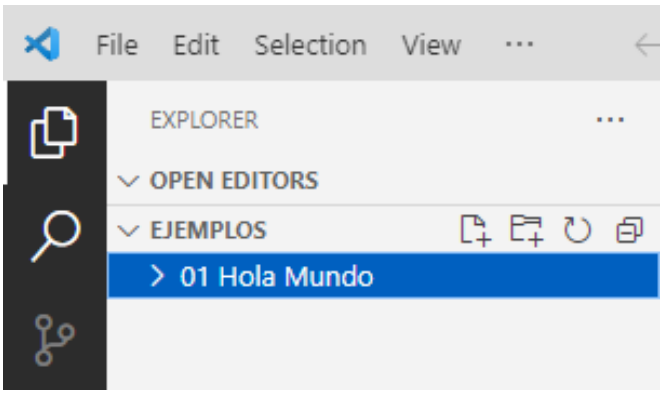
A la hora de trabajar con múltiples proyectos y no perdernos es importante seguir una jerarquía clara de carpetas. Por ese motivo recomiendo que, dentro de la carpeta que utilicéis para la asignatura, creéis una carpeta *U3* y dentro una subcarpeta *Ejemplos*.

A continuación, desde *Visual Studio Code* iremos a *File > Open Folder* y abriremos la carpeta *Ejemplos*. Se mostrará algo similar a lo que se puede ver en la siguiente imagen. La pestaña de bienvenida podemos cerrarla.

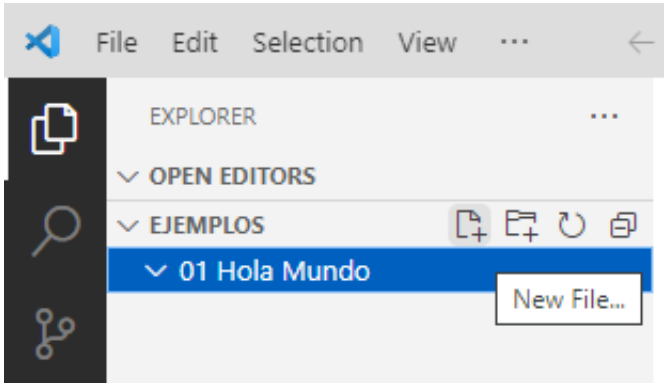


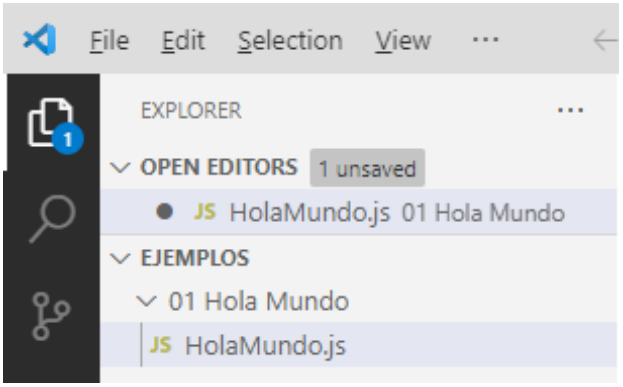
Cada proyecto se guardará en su correspondiente carpeta. Para crear nuestro primer proyecto pulsaremos en *New Folder* y lo llamaremos *01_hola_mundo*.





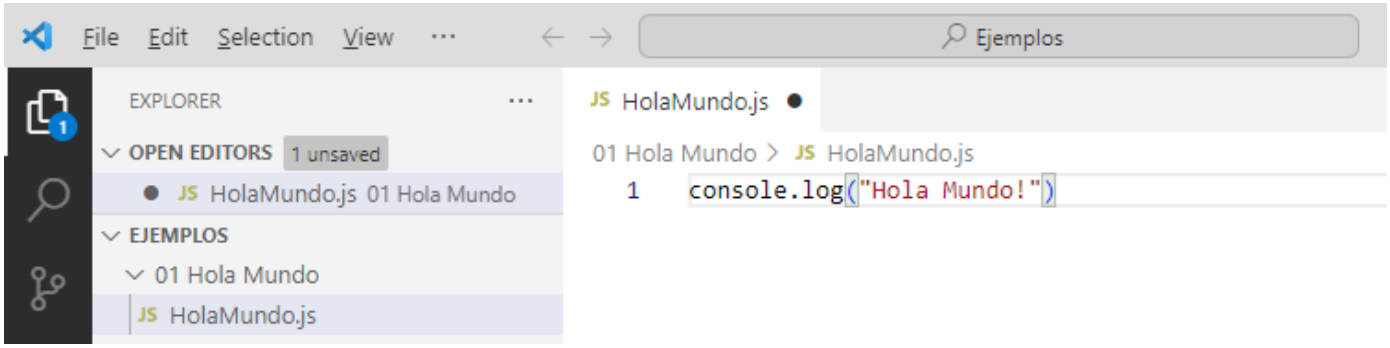
Dentro de la carpeta crearemos un nuevo archivo al que llamaremos *hola_mundo.js*.





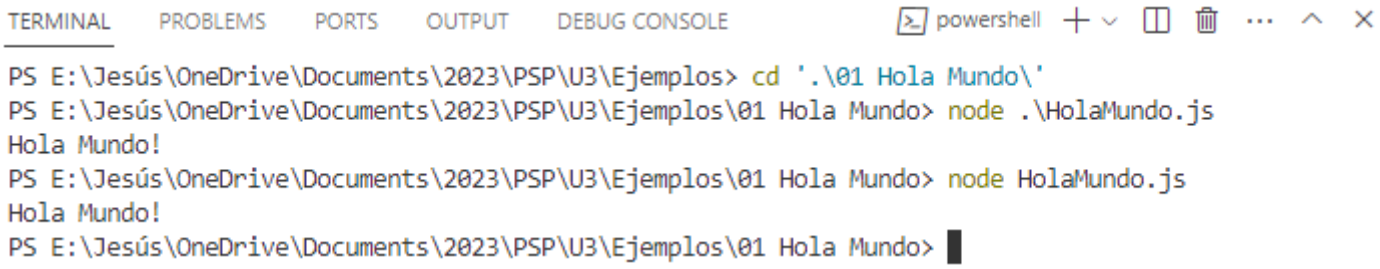
Al seleccionar el archivo se abrirá el editor, donde escribiremos la línea de código:

```
console.log("Hola mundo!")
```



El punto negro que se muestra en la pestaña del editor no sindica que no hemos guardado los cambios. Al pulsar *Ctrl + S* o *File > Save* el punto desaparecerá.

Para ejecutar nuestro programa abriremos la terminal de Visual Studio Code seleccionando el menú View > Terminal. En la terminal cambiaremos el directorio a la ruta de nuestro proyecto y lo ejecutaremos con el comando *node* como hicimos anteriormente.


TERMINAL
PROBLEMS
PORTS
OUTPUT
DEBUG CONSOLE
powershell
PS E:\Jesús\OneDrive\Documents\2023\PSP\U3\Ejemplos> cd '.\01 Hola Mundo\'
PS E:\Jesús\OneDrive\Documents\2023\PSP\U3\Ejemplos\01 Hola Mundo> node .\HolaMundo.js
Hola Mundo!
PS E:\Jesús\OneDrive\Documents\2023\PSP\U3\Ejemplos\01 Hola Mundo> node HolaMundo.js
Hola Mundo!
PS E:\Jesús\OneDrive\Documents\2023\PSP\U3\Ejemplos\01 Hola Mundo>

Módulos

En *Node.js* las funciones se agrupan en los denominados módulos. Los módulos no dejan de ser lo que en otros lenguajes llamaríamos librerías o bibliotecas. El uso de módulos permite reutilizar el código ya que podemos importarlas desde diferentes programas.

Módulos del *core*

El núcleo de Node.js incluye por defecto una serie de módulos que podemos utilizar en nuestro programa. Algunos de estos módulos son:

Módulo	Descripción	Documentación
fs	Permite trabajar con el sistema de archivos y directorios.	https://nodejs.org/api/fs.html
http	Permite crear un servidor HTTP	https://nodejs.org/api/http.html
https	Permite crear un servidor HTTPS	https://nodejs.org/api/https.html
net	Permite trabajar con sockets TCP	https://nodejs.org/api/net.html
os	Permite obtener información sobre el sistema operativo	https://nodejs.org/api/os.html
path	Permite trabajar con rutas de directorios y ficheros	https://nodejs.org/api/path.html
url	Permite *parsear* direcciones URL	https://nodejs.org/api/url.html
util	Ofrece múltiples utilidades como es el formateo de texto o *logging*.	https://nodejs.org/api/util.html

Importar módulos

Para importar un módulo utilizaremos la función *require* que recibe como parámetro el nombre del módulo a importar. Por ejemplo, para importar el módulo *fs* encargado de manipular el sistema de archivos escribiremos la siguiente línea.

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

Donde *fs* es una variable constante que utilizaremos para acceder a todas las funciones que ofrece el módulo.

Ejemplo 2 | Importación de módulos

Vamos a probar uno de los módulos que encontramos en el *core* de *Node.js*. En particular, probaremos el módulo *os* que nos permite obtener información relacionada con nuestro sistema. Para ello crearemos dentro de la carpeta *ejemplos* una nueva carpeta *"02_importacion"* y en su interior el archivo *importacion.js.*

En el archivo escribiremos las siguientes líneas:

```
const os = require ("os")

console.log(os.platform())
console.log(os.cpus())
```

En primer lugar, estamos importando el módulo *os* y posteriormente llamando a dos de sus métodos.

- *platform* devuelve un *string* que identifica la plataforma para la que Node.js está compilado. Puede devolver 'aix', 'darwin', 'freebsd','linux', 'openbsd', 'sunos', o 'win32'.
- *cpus* devuelve un *array* con información sobre los diferentes núcleos de nuestro microprocesador.

En https://nodejs.org/api/os.html podemos encontrar todas las funciones que nos ofrece el módulo *os*.

Creación de módulos propios

Por supuesto, es posible crear nuestros propios módulos, que también podremos importar con la función *require*.

Ejemplo 3 | Creación de módulos

Vamos a crear un nuevo módulo y a importarlo desde nuestro programa. Para ello crearemos dentro de la carpeta *ejemplos* una nueva carpeta *"03_mi_modulo"* y en su interior dos archivos *mi_modulo.js* y *principal.js.*

Dentro de *mi_modulo.js* incluiremos el siguiente código:

```
function suma (n1, n2){
    return n1 + n2;
}

function resta (n1, n2){
    return n1 - n2;
}

function multiplica (n1, n2){
    return n1 * n2;
}

module.exports = {
    add: suma,
    substract: resta
};
```

Podemos ver tres funciones muy sencillas: suma, resta y multiplica. Para que las funciones sean accesibles cuando importemos el módulo es necesario incluirlas en el objeto *module.exports*. Aquellas funciones que no incluyamos serán privadas.

Como podemos ver en el código anterior, podemos asignar un nuevo nombre a las funciones. Por ejemplo, la función *suma* se exporta como *add*, mientras que la función *resta* se exporta como *substract*.

Ahora vamos a importar nuestro módulo. Es tan sencillo como utilizar la función *require*, pero ahora tenemos que indicar la ruta relativa en la que se encuentra el archivo *js* que queremos importar. En *principal.js* incluiremos el siguiente código:

```
const mm = require("./mi_modulo.js");

let res = mm.add(4, 5);
console.log(res);
res = mm.substract(4, 5);
console.log(res);
```

Vemos que, a la hora de importar el módulo, estamos indicando con *./* que el archivo *mi_modulo.js* se encuentra en el mismo directorio que *principal.js*.

Módulos externos

Existe un gran número de módulos desarrollados por terceros que podemos incluir en nuestros proyectos. Estos módulos se agrupan en paquetes y son gestionados por *npm* (*Node Package Manager*). *npm* nos permite descargar e instalar estos paquetes en nuestros proyectos de forma sencilla, gestionando las posibles dependencias que tengan. En próximas sesiones aprenderemos a utilizar *npm*.

Ejemplos

Ejemplo 4a | Creación de un servidor HTTP

Con los módulos que encontramos en el *core* de *Node.js* podemos hacer algunas cosas muy interesantes. En primer lugar, vamos a crear un servidor con el módulo *http*. Para ello crearemos dentro de la carpeta *ejemplos* una nueva carpeta *"04a_servidor_http"* y en su interior el archivo *servidor_http.js.*

En la primera línea importaremos el módulo http del siguiente modo:

```
const http = require("http")
```

Si consultamos la documentación veremos que existe un método *createServer* que, como su nombre indica, nos permite crear un servidor.

http.createServer([options][, requestListener])

▶ History

- `options` <Object>
- `requestListener` <Function>
- Returns: <http.Server>

Returns a new instance of `http.Server`.

The `requestListener` is a function which is automatically added to the `'request'` event.

El método tiene dos parámetros de entrada que son opcionales:

- *options*, que es de tipo *Object*. Se trata de un objeto donde podemos indicar opciones de configuración del servidor. Ignoraremos este parámetro.
- *requestListener*, es de tipo *Function*. Es la función que se ejecutará cuando se produzca una petición al servidor o, como indica la documentación, cuando se produzca un evento *request*. Dicho de otro modo, nos permite indicar qué función queremos que se ejecute cada vez que se produzca una petición. Pero ¿Cómo debe ser esta función? ¿Tiene que recibir algún parámetro?

Si consultamos la documentación del evento *request* veremos que tiene dos objetos asociados: *request* y response.

Event: 'request'

Added in: v0.1.0

- `request` <http.IncomingMessage>
- `response` <http.ServerResponse>

Emitted each time there is a request. There may be multiple requests per connection (in the case of HTTP Keep-Alive connections).

- *request* es una instancia de *http.ServerRequest* y contendrá toda la información relativa a la petición que se ha realizado.
- *response* es una instancia de *http.ServerResponse* y es el objeto que nos permitirá responder a la petición.

Así pues, si queremos pasar un *requestListener* para que se ejecute cuando se produzca un evento *request*, tendremos que implementar una función con dos parámetros. Esto lo haremos con el siguiente código:

```
const requestListener = function (request, response){
    console.log("Se ha producido una petición")
};
```

Hay varias cosas que nos debe llamar la atención del código anterior:

- La primera es que estamos guardando una función en una variable. Esto es algo muy común en *JavaScript* y es una característica de los lenguajes funcionales.
- La variable se llama `requestListener` pero podría tener cualquier otro nombre. Se ha puesto este nombre para que el código sea más claro y coincida con el nombre del parámetro de la función *createServer*.
- Del mismo modo, el nombre de las variables *request* y *response* podría ser cualquier otro.
- En ningún momento se indican los tipos de las variables. Se dice que *JavaScript* es un lenguaje débilmente tipado.
- La palabra clave *const* sirve para indicar que la variable *requestListener* es constante y que, por tanto, no podrá modificarse o volver a declararse.

Ahora ya podemos llamar a la función *createServer* del siguiente modo:

```
const server = http.createServer(requestListener);
```

Podemos ver que a la función *createServer* le estamos pasando la variable *requestListener* como parámetro. Esta variable contiene la función que hemos definido anteriormente. La llamada *createServer* devuelve un objeto de tipo *Server* que estamos guardando en la variable *server*.

Por último, tenemos que poner en marcha nuestro servidor. Esto es tan sencillo como llamar a la función *listen* del objeto *server*.

```
server.listen(80);
```

Esto hará que el servidor espere conexiones en el puerto 80 de nuestra máquina (*localhost)*.

Si ejecutamos nuestro programa e introducimos la dirección *localhost* o 127.0.0.1 en cualquier navegador (*Chrome*, *Edge*, *Firefox*, etc.), veremos que en el terminal de *Visual Studio Code* se mostrará el mensaje “Se ha producido una petición”.

```
PS E:\Jesús\OneDrive\Documents\2023\PSP\U3\Ejemplos\02_servidor_http> node servidor_http.js
Se ha producido una petición
Se ha producido una petición
```

Ejemplo 4b | Cabeceras HTTP

Partiendo del ejemplo anterior vamos a ver cómo obtener los campos incluidos en la cabecera de las peticiones HTTP que recibe nuestro servidor. Para realizar este ejemplo haremos una copia de *"04a_servidor_http"* y la renombraremos como *"04b_cabeceras_http"*.

Podemos obtener los campos de una petición a través del objeto *request*. Para comprobarlo, modificaremos la función *requestListener* con el siguiente código:

```
const requestListener = function (request, response){
    console.log("Se ha producido una petición")
    console.log("URL: " + request.url)
    console.log("Método: " + request.method)
    console.log("Algunos campos de la cabecera:")
    console.log("Host: " + request.headers['host'])
    console.log("User-Agent: " + request.headers['user-agent'])
    console.log("Accept-Language: " + request.headers['accept-language'])
};
```

Si ejecutamos de nuevo el programa y lanzamos una petición desde nuestro navegador, veremos que en la terminal se mostrará la URL del recurso que se ha solicitado, el método utilizado (GET) y algunos de los campos de las cabeceras. En particular, en este caso mostramos el *host*, el *user-agent* y *accept-language*.

Por ejemplo, si abrimos el navegador y accedemos a http://*localhost/psp/test* obtendremos un resultado similar al siguiente:

```
Se ha producido una petición
URL: /psp/test
Método: GET
Algunos campos de la cabecera:
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/119.0.0.0 Safari/537.36
Accept-Language: es,en;q=0.9
```

Ejemplo 4c | Respondiendo a peticiones

En este ejemplo vamos a ver cómo podemos responder a las peticiones que recibe nuestro servidor. Para realizar este ejemplo haremos una copia de *"04b_servidor_http"* y la renombraremos como *"04c_respuestas_http"*.

Dentro de nuestra función *requestListener* incluiremos las siguientes líneas:

```
response.setHeader("Content-Type", "text/html");
response.writeHead(200);
response.write("<h1>Hola mundo!</h1>");
response.end();
```

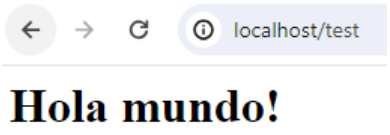
El objeto *response* es el que nos permite responder a las peticiones. Mediante su función *setHeader* podemos añadir los campos que queramos a la cabecera de nuestra respuesta. *writeHead* se encarga de enviar la cabecera con el código de estado que indiquemos. Podemos utilizar la propia llamada *writeHead* para establecer los campos de la cabecera como se muestra a continuación:

```
response.writeHead(200,
    {'Content-Length': "text/html"});
```

El método *write* permite enviar datos en el cuerpo de la respuesta, podemos enviar cadenas de texto o *bytes*. Finalmente, la función *end* permite indicar que ya hemos completado la respuesta. Podemos utilizar la propia llamada *end* para enviar datos.

```
response.end("<h1>Hola mundo!</h1>");
```

Si accedemos ahora a nuestro servidor veremos que muestra el mensaje "Hola mundo!" con el estilo "*h1*". Nuestro servidor está respondiendo a las peticiones con una página *html*.



Ejemplo 5 | Lectura de ficheros

En este ejemplo vamos a ver cómo utilizar el módulo *fs* para leer archivos. Para ello crearemos dentro de la carpeta *ejemplos* una nueva carpeta "*05_lectura_ficheros*" y en su interior crearemos dos archivos: *lectura_ficheros.js* que contendrá el código fuente de nuestro programa y *prueba.txt*, que será el archivo que leeremos.

Siempre es interesante que visitemos la documentación oficial para familiarizarnos con su uso. En el siguiente enlace podemos ver que el módulo *fs* dispone de un método *readFile*.

https://nodejs.org/dist/latest-v20.x/docs/api/fs.html#fsreadfilepath-options-callback

fs.readFile(path[, options], callback)

▶ History

- `path` `<string>` | `<Buffer>` | `<URL>` | `<integer>` filename or file descriptor
- `options` `<Object>` | `<string>`
  - `encoding` `<string>` | `<null>` **Default:** `null`
  - `flag` `<string>` See support of file system `flags`. **Default:** `'r'`.
  - `signal` `<AbortSignal>` allows aborting an in-progress readFile
- `callback` `<Function>`
  - `err` `<Error>` | `<AggregateError>`
  - `data` `<string>` | `<Buffer>`

Asynchronously reads the entire contents of a file.

Dicha función recibe por parámetro el *path* del fichero, opciones que permiten indicar aspectos como la codificación del archivo y por último una función *callback*. Esta función se llamará cuando se finalice la lectura del fichero y recibe dos parámetros *err* y *data*. El primer parámetro, de tipo *Error*, tendrá información sobre un posible error que se haya podido producir y *data* contendrá los datos leídos.

Así pues, podríamos implementar la lectura del archivo del siguiente modo:

```
const fs = require("fs")

const lectura = function(err, data){
    if (err) {
        console.error(err);
        return;
    }
    console.log(data);
}

fs.readFile('./prueba.txt', lectura);
```

Aprovechamos este ejercicio para introducir las expresiones en lambda en *JavaScript*. Su notación es muy similar a la de Java, pero en vez de utilizar la flecha -> se utiliza =>. Así pues, el código anterior se podría escribir como:

```
const fs = require("fs")

const lectura = (err, data) => {
    if (err) {
        console.error(err);
        return;
    }
    console.log(data);
}

fs.readFile('./prueba.txt', lectura);
```

E incluso podríamos omitir la creación de la variable lectura y poner la expresión *lambda* directamente en el parámetro de la función *readFile* como se muestra a continuación:

```
const fs = require("fs")

fs.readFile('./prueba.txt', (err, data) => {
    if (err) {
        console.error(err);
        return;
    }
    console.log(data);
});
```

Práctica

Parte obligatoria

En esta práctica vamos a desarrollar un pequeño servidor con *Node.js* y el módulo *http*. El servidor nos permitirá acceder a una página web que también tendrás que desarrollar. Para realizar este proyecto crearemos una carpeta *Práctica1* donde guardaremos todos los archivos. El archivo con el código *JavaScript* se llamará *practica1.js*.

- La página web a desarrollar podrá ser de la temática que queramos, será muy sencilla y será suficiente con que tenga:
    - 3 archivos *html* (uno de ellos será *index.html*): Mediante enlaces se tendrá que poder navegar a todas las páginas. Los textos pueden ser copiados/pegados de Internet o generados mediante una IA.
    - 1 archivo *css*: Es suficiente con que aplique estilo a una etiqueta. Por ejemplo, que modifique el estilo de las cabeceras *h1*. Al menos *index.html* debe importar el archivo *css* haciendo uso de la etiqueta *link*.
    - 1 imagen *jpg* que también se incluirá en *index.html.*
- Nuestro servidor *node.js* tendrá que obtener el recurso (URL) sobre el que se hace la petición. Como respuesta devolverá los datos del recurso correspondiente o, dicho de otro modo, el contenido del fichero correspondiente. Por ejemplo, si desde el navegador accedemos a la URL *localhost*, *node.js* responderá con el recurso *index.html*. Si accedemos a la URL *localhost/pepito.html*, *node.js* responderá con el contenido del archivo *pepito.html*.
- En caso de que el recurso no exista, *node.js* devolverá el código de error que corresponda según el protocolo HTTP.
- La respuesta del servidor deberá incluir el campo *Content-Type* en la cabecera con el valor adecuado. Para realizar esta funcionalidad crearemos un módulo propio con una función que recibirá por parámetro un *string* con la extensión de un archivo y devolverá el *Content-Type* correspondiente. Por ejemplo, si la función recibe el parámetro "*.html*" devolverá "*text/html*". Si la función recibe el parámetro "*.css*" devolverá "*text/css*", etc.
- Para extraer la extensión a partir de un nombre de archivo utiliza el módulo *path* y su método *extname*.

Consejos

- No te dediques en primer lugar a hacer toda la página web. Siempre hay que empezar por resolver el problema más complejo y/o el más importante. Os recomendaría que siguierais estos pasos:
    - Es buena idea empezar creando solo con un archivo *index.html* que no incluya imágenes ni estilos, e intentar que la página se muestre en nuestro navegador al acceder a *localhost*.

    - Una vez consigas esto puedes probar a crear las otras dos páginas y añadir los enlaces entre ellas. Comprueba que sigue funcionando y en caso afirmativo, prueba a añadir la imagen o imágenes y los estilos.
    - Si todo ha ido bien, implementa el módulo con la función encargada de obtener el *Content-Type*.
- Recordad que a la hora de escribir rutas (para los enlaces, incluir el *css* o la imagen) lo mejor es utilizar rutas relativas.
- Esta práctica se puede resolver con todo lo que hemos visto en este documento. No es necesario utilizar librerías externas o código rebuscado.
- Para facilitar la tarea es recomendable que todos los archivos se ubiquen en el directorio raíz (*Práctica1*), es decir, que no se utilicen subdirectorios.

Parte opcional

Redirecciones en HTTP

En el protocolo HTTP, una redirección es una respuesta del servidor que indica al cliente que la URL solicitada ha sido movida a otra ubicación.

Para simular este cambio de ubicación cambiaremos el nombre del archivo *css*. Por ejemplo, si nuestro archivo *css* se llama *styles.css* podríamos cambiarle el nombre a *estilos.css*. ¡Ojo! Solo cambiamos el nombre del archivo, pero no modificamos las referencias que existan en los *html*.

En el caso de que el servidor reciba una petición para obtener el recurso *styles.css* deberá responder que el recurso se ha movido a otra ubicación y deberá indicar al cliente la nueva URL.

Query string

El *query string* es una forma común de enviar datos desde el cliente al servidor a través de las URL y se emplea ampliamente en aplicaciones web para transmitir parámetros, realizar consultas o personalizar la experiencia del usuario. Es la parte de la dirección web que sigue al signo de interrogación "?" y contiene datos adicionales enviados al servidor web.

Por ejemplo, dada la siguiente URL:

```
http://localhost/industria?lang=es&color=black
```

El *query string* contiene dos pares clave-valor: *lang=es* y *color=black.*

Se propone modificar el proyecto para que al recibir el parámetro *lang=en* el servidor devuelva una versión de la página web en inglés. Para ello realiza una copia de los archivos *.html* de forma que sus nombres empiecen por *en*. Por ejemplo, si los archivos *html* se llaman *index.html*, *industria.html* y *proceso.html*, las copias se llamarán *enindex.html*, *enindustria.html* y *enproceso.html*. No es necesario traducir el contenido de los archivos y es suficiente con añadir alguna señal (por ejemplo, las letras EN) en el contenido de la página para poder diferenciarlos con la versión en español.

Pistas

El módulo *url* permite obtener y separar las partes de una URL.

```
var url = require('url');
var url_parts = url.parse(request.url, true);
console.log(url_parts);
```

Si el parámetro *lang* no existe dentro del *query string* no habrá que hacer nada, pero si existe, habrá que concatenar su valor (en) al nombre del recurso solicitado. Por ejemplo, si accedemos a la URL

http://localhost/proceso.html?lang=en

El servidor, en vez de leer el archivo *./proceso.html*, deberá leer el archivo *./enproceso.html*.