



# Curso de Back-End con Node.js (Inicial)

## Clase 02



# Temario



# Temario

- Mostrar todos los archivos y todas las extensiones.
- Línea de comandos.
- Node.js:
  - ¿Qué es?
  - Instalación.
  - ¿Por qué usar Node.js?
  - Módulos en Node.
  - Event Loop.



Mostrar todos los archivos  
y todas las extensiones



# Mostrar todos los archivos y todas las extensiones

Es muy aconsejable que sigan los siguientes pasos para poder ver todos los archivos **ocultos** en su sistema operativo, así como las **extensiones** de todos los archivos.

En Windows:

- [Mostrar archivos ocultos.](#)
- [Mostrar todas las extensiones.](#)

En Mac:

- [Mostrar archivos ocultos.](#)
- [Mostrar todas las extensiones.](#)



# Línea de comandos



# Línea de comandos

- Si bien no son sinónimos, se suele hablar de lo mismo al usar los términos: línea de comandos, consola, terminal, shell, bash, etc. Más info [aquí](#).
- En general, estos términos hacen referencia al software que permite **interactuar con una computadora** mediante comandos ingresados con **líneas de texto** (seguidos de un *Enter*). El término más correcto en este caso es el de “terminal”.
- El uso de una **terminal** permite tener más control sobre un equipo y realizar tareas de forma más rápida (una vez que le agarran la mano).
- ¿Cómo abrir la terminal?
  - **Mac:** Presionar botón Command (CMD) + Espacio. Luego escribir “Terminal”.
  - **Windows:** Abrir buscador, escribir “cmd” y presionar Enter.
  - 🖱️ También hay una terminal integrada en **Visual Studio Code**.



# Terminal – Lenguajes (o shells)

Así como existen distintos lenguajes de programación con los cuales se pueden escribir programas, también existen distintos **lenguajes** (o **shells**) para escribir comandos dentro de una terminal. Ejemplos:

- Shell (`sh`) – también llamado “Bourne Shell”.
- Bash (`bash`) – también llamado “Bourne-Again Shell”.
- Z-Shell (`zsh`).
- Command shell – también llamada “CMD”. Viene por defecto en Windows.
- PowerShell (`pwsh`).





# Terminal vs. Shell – Resumen

- **Terminal** es el software (la ventana) donde se van a escribir los comandos de texto.
- Cada sistema operativo trae una terminal por defecto, pero se pueden instalar otras como [iTerm](#) o [Hyper](#). Además, [Visual Studio Code](#) también trae su propia terminal.
- Dentro de dicha terminal, los comandos se pueden escribir de una forma u otra según el lenguaje (intérprete) que queramos usar, es decir, la **shell** que queramos usar.
- Es decir, podríamos usar la terminal **Hyper** junto con la shell **Bash**, o podríamos usar la terminal **Hyper** junto con la shell **Zsh**. También se podría usar junto con **PowerShell**.



# Terminal – Algunos comandos comunes

👉 Recordar que la sintaxis depende de la shell (lenguaje) que estamos usando.

	En Unix (Mac, Linux) – Shell: Bash	En Windows – Shell: CMD
Conocer el directorio actual	<code>pwd</code>	<code>cd</code>
Listar archivos y directorios dentro del directorio actual	<code>ls</code>	<code>dir</code>
Cambiar de directorio	<code>cd /ruta/a/CarpetaDestino</code> <code>cd ..</code>	<code>cd /ruta/a/CarpetaDestino</code> <code>cd ..</code>
Borrar la pantalla	<code>clear</code>	<code>cls</code>
Otros comandos	<a href="http://www.yolinux.com/TUTORIALS/unix_for_dos_users.html">http://www.yolinux.com/TUTORIALS/unix_for_dos_users.html</a>	



Last login: Sat May 20 22:51:11 on ttys003

You have new mail.

Marcuss-MacBook-Pro:~ Marcus\$

Este texto es lo que se conoce como “**prompt**” e indica el comienzo de un comando en la terminal.

Por defecto, en Mac y Linux, el prompt termina con el símbolo \$ (pesos) mientras que en Windows se usa el símbolo > (mayor).

El prompt puede ser configurado para que sea otro texto que queramos.



Tengan cuidado al usar la terminal. Si meten la pata pueden causar daños irreversibles (como borrar un montón de archivos para siempre).



# Terminal – Hyper

(Instalación opcional)

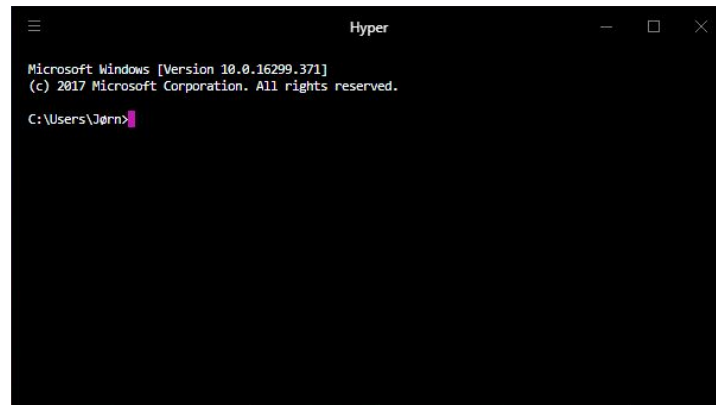


```
▲ ~/ # Hyper is an Electron-based Terminal
▲ ~/ # Built on HTML/CSS/JS
▲ ~/ # Fully extensible!
▲ ~/ █
```



# Terminal – Hyper

- Sería bastante útil que todos tengamos la misma terminal, sin importar nuestro sistema operativo (Windows, Linux o Mac). De hecho, si todos usamos VSC, ya estaremos usando la misma terminal.
- Además, si lo desean, pueden instalar una terminal llamada Hyper.  
Link: <https://hyper.is>.
- Recuerden que esto sólo hará que todos tengamos la misma **terminal** (la misma ventana), pero aún seguimos sin tener el mismo intérprete de comando (seguimos sin tener el mismo **shell**).
- Para modificar el shell en Windows, ver [este artículo](#) o [este otro](#).



Así se vería Hyper en **Windows**, con la **shell** que viene por defecto con el sistema operativo.



# Windows Subsystem for Linux (WSL)





# Windows Subsystem for Linux (WSL)

- Los usuarios de **Windows 10**, si lo desean, pueden probar algo llamado Windows Subsystem for Linux (WSL).
- Permite utilizar Linux dentro de Windows 🤖. Es una especie de máquina virtual, pero comparten el mismo filesystem entre Windows y Linux.
- Gracias a WSL, se pueden correr comandos y programas de Linux, pero dentro de Windows. Por ejemplo, puede acceder a una terminal de Linux donde se pueden ejecutar comandos Bash.
- Video tutorial: <https://www.youtube.com/watch?v=xzgwDbe7foQ>.



# Tareas extra



# Tareas extra

Investigar sobre los siguientes comandos:

- `sudo` → ¿qué es? ¿para qué sirve? ¿cuándo usarlo?
- `rm -r`
- `df`
- `cat`
- `head`
- `tail`
- `curl`
- `grep`



¿Qué es Node.js?



# ¿Qué es Node.js? (1/3)



*“Es un runtime que permite ejecutar  
**JavaScript** fuera de los navegadores”.*



## ¿Qué es Node.js? (2/3)

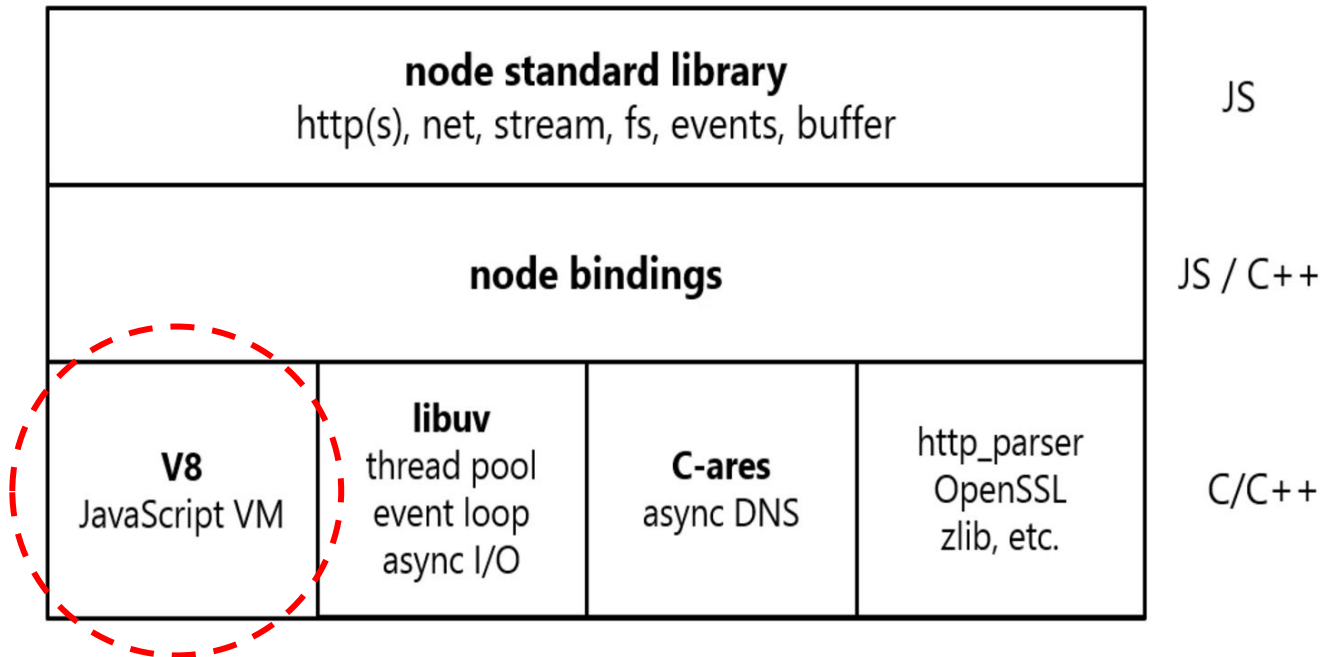
Node.js es básicamente un **runtime** (construido sobre el motor de JavaScript V8 de Google Chrome) que permite **ejecutar JavaScript** fuera de los navegadores.

Es una herramienta que permite construir **servidores web** y aplicaciones de red **escalables y asincrónicas**.

Está escrito en C++ (mayormente) y JavaScript.



# ¿Qué es Node.js? (3/3) – Arquitectura



Fuente: Amazing Features of Node.js that makes it in top 5 Server Side Scripts

Disponible es: <http://www.justtotaltech.co.uk/blog/amazing-features-node-js-top-5-server-side-scripts>.



# Instalación de Node





# Instalación

Deberán instalar **Node.js** de <https://nodejs.org>. Seleccionar versión **LTS (v14)**.

La instalación incluye el gestor de dependencias **npm** – <https://www.npmjs.com>, aunque luego, si lo desean, podrán usar otro gestor de dependencias como Yarn.

👉 Ya hablaremos en más en detalle sobre esto.

---

Además, precisarán un editor de código. Recomendamos utilizar **Visual Studio Code**.

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



# Hola Mundo (básico)



# Hola Mundo (básico)

En un archivo `index.js` escribir:

```
console.log("Hola Mundo");
```

Luego, en una terminal ejecutar:

```
node index.js
```

En la terminal aparecerá el texto "Hola Mundo".



# Aclaración

Vale la pena aclarar que el código JavaScript que escribimos en Node.js corre en un servidor, **no en un navegador**.

Por lo tanto, no están disponibles los objetos `document` ni `window`, y por lo tanto no se pueden escribir código como:

```
const parrafo = document.querySelector("p");
```

```
window.alert(";Hola Mundo!");
```

Por otro lado, con Node.js se podrá escribir código que no puede correr un navegador, como el que permite acceder al File System de una computadora.



# Ejecutar JavaScript directamente en la terminal



# Ejecutar JavaScript directamente en la terminal

Si en una terminal escriben solamente “node” y le dan Enter, les aparecerá una línea de comandos donde podrán ingresar y ejecutar código JavaScript desde ahí mismo, muy similar a la consola de un navegador.



```
→ ~ node
Welcome to Node.js v13.11.0.
Type ".help" for more information.
> var nombre = "María";
undefined
> console.log(nombre);
María
undefined
> 5 * 9
45
> █
```

Para salir deberán escribir ".exit" o sino CTRL + C.



¿Por qué Node.js?



# ¿Por qué Node.js? (1/4)

A nivel de Back-End:

*“I/O debe hacerse diferente. Lo hemos estado haciendo mal durante años”.*

Ryan Dahl, 2009.





## ¿Por qué Node.js? (2/4)

Antiguamente, los servidores web funcionaban haciendo consultas similares al siguiente ejemplo:

1. **Consultar** una base de datos.  
Esperar por el resultado, la ejecución del programa queda congelada.
2. **Recibir** un resultado de la base de datos.
3. **Usar** el resultado obtenido.

```
var result = db.query("SELECT * FROM..."); // Zzzzz.  
// Se usa el resultado.
```



# ¿Por qué Node.js? (3/4) – Event Driven

Volviendo al ejemplo de la consulta a la base de datos, Node.js propone resolverlo utilizando un sólo *thread* (hilo) y un [event loop](#). Los pasos a seguir serían:

1. **Consultar** la base de datos.
2. **No esperar** por el resultado, pero “estar atento” a que el mismo esté pronto. Mientras tanto, realizar otras tareas.
3. Cuando la base de datos tenga pronto el resultado y lo retorne (en el momento que sea), **hacer** algo con el mismo.



# ¿Por qué Node.js? (4/4) – Event Driven

El ejemplo anterior en Node.js *sería* algo así:

```
db.query("SELECT * FROM...", function() {  
    // Se usa el resultado.  
});
```

👉 También pueden ver [este video](#) que explica muy bien porqué usar Node.js.



# Módulos de Node

# Módulos de Node

Luego usaremos **npm** para instalar otros módulos (externos).



Node cuenta con varios “módulos” (especie de librerías JavaScript) integrados que permiten realizar distintas tareas como, por ejemplo:

- Acceder al **filesystem** (disco duro). Esto permite crear, modificar y eliminar archivos y carpetas.
- Crear un servidor **http**.
- Gestionar **eventos**.
- Parsear **urls**.
- Utilizar funciones **criptográficas**.
- Etc. La lista completa se puede acceder aquí: <https://nodejs.org/api>.



# HTTP

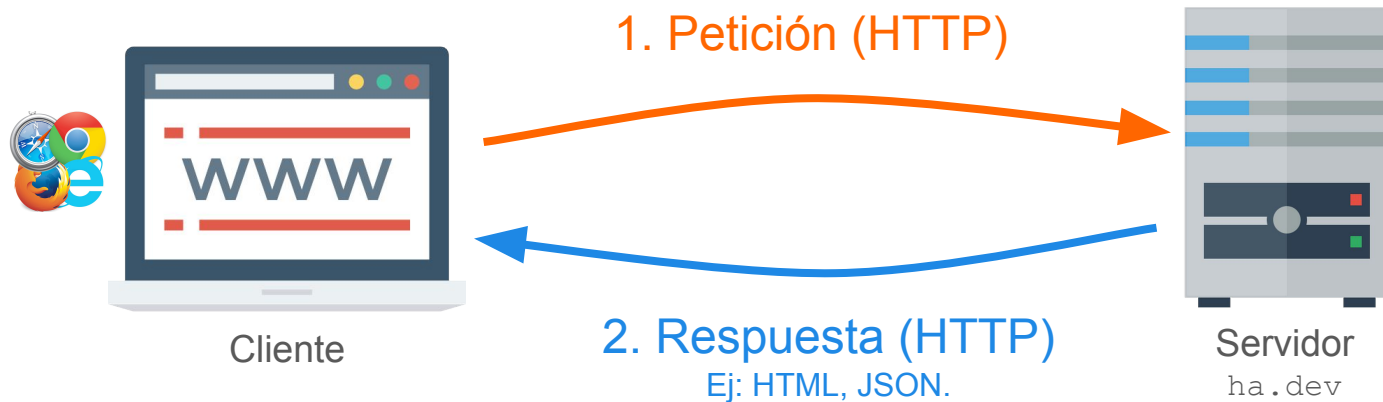
# Hypertext Transfer Protocol (HTTP) (1/3)



1. Cuando un usuario escribe una URL en su navegador y presiona `Enter`, lo que hace es realizarle una **petición** o **request** (HTTP) a un servidor web. Técnicamente esto se conoce como un request de tipo GET (lo veremos en breve).

Básicamente el usuario le dice al servidor: *“Dame lo que haya en esta URL: `https://ha.dev`”*.

# Hypertext Transfer Protocol (HTTP) (2/3)



2. Al recibir la petición, una aplicación en el servidor la analiza, la procesa y responde al cliente con una **respuesta** o **response** que suele ser en formato HTML (pero puede ser JSON, JavaScript, CSS, una imagen o texto plano). Los posibles formatos se pueden consultar [aquí](#).

El servidor jamás retorna código PHP, Java, Ruby, Python, C# u otro código de Back-End. Recordar que el navegador sólo "entiende" cosas como HTML, CSS, JavaScript e imágenes.





# Hypertext Transfer Protocol (HTTP) (3/3)

La comunicación anterior, entre el cliente y el servidor, se realiza utilizando un **protocolo** llamado **HTTP** o **HTTPS**.

Un protocolo es un **conjunto de reglas** que indican cómo debe ser la comunicación entre dos equipos, estableciendo la forma de identificación de los equipos en la red, la forma de transmisión de los datos y la forma en que la información debe procesarse.

La comunicación con un sitio web puede realizarse usando el protocolo HTTP (sin encriptar) o el HTTPS (encriptado).

Otros protocolos que tal vez conocen son: FTP, SMTP, IMAP, POP, SSH, TCP, UDP, etc.



# Estructura de una URL



# Estructura de una URL (1/2)

`https://ha.dev:443/productos`



Protocolo



Dominio



Puerto




Ruta (path)

- **Protocolo:** Conjunto de reglas que indican cómo debe ser la comunicación entre dos equipos, estableciendo la forma de identificación de los equipos en la red, la forma de transmisión de los datos y la forma en que la información debe procesarse. Por defecto se usa el protocolo `HTTP`, pero se podría usar `HTTPS`.
- **Dominio:** Dirección (string) que permite acceder de forma amigable a la dirección IP donde se encuentra el servidor. Ej: `ha.dev` apunta a la IP `76.76.21.21`.
- **Puerto:** Interfaz de comunicación en un sistema operativo. No es hardware, es un concepto lógico (software). Por defecto los servidores web se encuentran “escuchando” en el puerto 80 para `HTTP` y el puerto 443 para `HTTPS`.
- **Ruta:** El recurso que se le está pidiendo al servidor. Podría ser un archivo como `avion.jpg`.



## Estructura de una URL (2/2)

`https://ha.dev:443/productos?moneda=USD`



Query String

- **Query String:** En la URL también se puede agregar datos adicionales que serán enviados al servidor a través de un conjunto de parámetros llamado *query string*.
  - El símbolo de pregunta ? marca el comienzo del *query string*.
  - En este ejemplo, el *query string* contiene un sólo parámetro llamado `moneda` con el valor `USD`. En este caso, esta información adicional es para avisarle al servidor que muestre los precios de los productos en dólares.
  - Se pueden enviar varios parámetros, separándolos con un ampersand &.



Hola Mundo (vía http)



# Hola Mundo (vía http) (1/2)

A continuación se hará uso del módulo `http`, el cual permite crear un pequeño servidor que estará escuchando peticiones en el puerto 3000.

En un archivo `index.js`:

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

const server = http.createServer(function(req, res) {
  res.end("Hola Mundo");
});

server.listen(3000);
```

`require` es una función de Node que se utiliza para importar módulos.



# Hola Mundo (vía http) (2/2)

Luego, en una terminal ejecutar:

```
node index.js
```

Notar que la terminal quedará como “paralizada”. Esto no es un error, es simplemente el **servidor** que está corriendo, esperando por peticiones. Si se cierra la terminal, se cerrará el servidor.

👉 Finalmente, entrar a través de un navegador a <http://localhost:3000>.



# Event Loop





# Event Loop

El Event Loop es lo que **permite que Node.js utilice un único *thread*** (hilo). Para entenderlo, conviene primero entender el paradigma de programación dirigida por eventos (*Event-Driven Programming*), que hace años viene siendo utilizado en programación de Front-End.

El *event-driven programming* es **paradigma de programación orientado a eventos**. Ejemplos de este tipo de programación: Realizar una tarea cuando se obtuvo un registro de una base de datos, se completó una petición (*request*), o se leyó un archivo del *file-system*.

Un entorno como el de Node permite mediante un mecanismo central que se encuentra permanentemente escuchando eventos, llamar funciones (llamadas “*callback*”) cuando el evento es detectado.

👉 Recomendamos leer [este artículo](#), [este video](#) y [este “juego”](#).



Entonces, ¿qué es Node.js?



# Entonces, ¿qué es Node.js?

*“Node.js es una infraestructura **no-bloqueante** y **orientada a eventos** para desarrollar programas con **alta concurrencia**.”*

Los programadores que utilicen Node.js pueden despreocuparse de la ocurrencia de *deadlocks*.



# Ejercicio 1



# Ejercicio 1 (1/4)

A continuación se hará el ejemplo del “Hola Mundo!” (que se vio anteriormente) y se lo analizará por partes.

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

const server = http.createServer((req, res) => {
  res.end("Hola Mundo!\n");
});

server.listen(8080, () => {
  console.log("Servidor escuchando en http://localhost:8080.");
});
```

Esta es una nueva forma de definir funciones, llamada: **Arrow Functions**. Ver más [aquí](#).



# Ejercicio 1 (2/4)

Primero se importa el módulo integrado `http`, y se lo asigna a una variable con el mismo nombre (aunque podría ser otro).

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

const server = http.createServer((req, res) => {
  res.end("Hola Mundo!\n");
});

server.listen(8080, () => {
  console.log("Servidor escuchando en http://localhost:8080.");
});
```



# Ejercicio 1 (3/4)

Luego se “instancia” un nuevo servidor HTTP pasándole una función *callback* que contestará a cada *request* el mensaje “Hola Mundo”.

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

const server = http.createServer((req, res) => {
  res.end("Hola Mundo!\n");
});

server.listen(8080, () => {
  console.log("Servidor escuchando en http://localhost:8080.");
});
```



# Ejercicio 1 (4/4)

Finalmente se invoca la función `listen()` pasándole un puerto y una función *callback* que se ejecutará una vez que el servidor esté listo para recibir y contestar *requests*.

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

const server = http.createServer((req, res) => {
  res.end("Hola Mundo!\n");
});

server.listen(8080, () => {
  console.log("Servidor escuchando en http://localhost:8080.");
});
```





# Ejercicio 2



## Ejercicio 2 (1/2)

Se reutilizará el siguiente código (del ejercicio anterior):

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

const server = http.createServer((req, res) => {
  // Código que se ejecutará cuando el servidor reciba un request.
});

server.listen(8081, () => {
  console.log("Servidor escuchando en http://localhost:8081.");
});
```



## Ejercicio 2 (2/2)

Importar el módulo [File System](#) `fs` y utilizar la función `fs.writeFile` para crear o recrear un archivo de texto toda vez que el servidor recibe una requisición HTTP.

```
var http = require("http");
var fs = require("fs");

var server = http.createServer((req, res) => {
  fs.writeFile("archivo.txt", "Hola Mundo!\n", (err) => {
    if (err) return res.end("Ha ocurrido un error.");
    res.end("Se ha creado un archivo!\n");
  });
});

server.listen(8081, () => {
  console.log("Servidor escuchando en http://localhost:8081.");
});
```



# Ejercicio 3



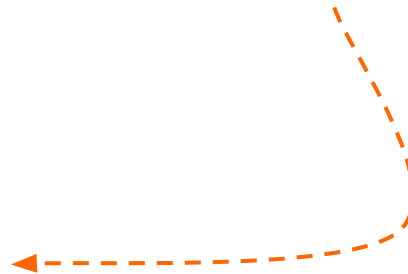
## Ejercicio 3 (1/3)

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

const server = http.createServer((req, res) => {
  fs.writeFile("archivo.txt", "Hola Mundo!\n", (err) => {
    if (err) return res.end("Ha ocurrido un error.");
    res.end("Se ha creado un archivo!\n");
  });
});

server.listen(8082, () => {
  console.log("Servidor escuchando en http://localhost:8082.");
});
```

Usar como base el código del ejercicio anterior.





## Ejercicio 3 (2/3)

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

const server = http.createServer((req, res) => {
  fs.appendFile("archivo.txt", "Hola Mundo!\n", (err) => {
    if (err) return res.end("Ha ocurrido un error.");
    res.end("Se ha creado un archivo!\n");
  });
});

server.listen(8082, () => {
  console.log("Servidor escuchando en http://localhost:8082.");
});
```

Utilizar la función `fs.appendFile` para agregar una nueva línea de texto al archivo toda vez que el servidor recibe un request HTTP.

🤔 ¿Se les ocurre alguna idea de cómo asegurarnos que cada nueva entrada (cada nueva línea de texto) tenga un contenido distinto?



## Ejercicio 3 (3/3)

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

let count = 0;

const server = http.createServer((req, res) => {
  fs.appendFile("archivo.txt", ++count + "\n", (err) => {
    if (err) return res.end("Ha ocurrido un error.");
    res.end("Se ha creado un archivo!\n");
  });
});

server.listen(8082, () => {
  console.log("Servidor escuchando en http://localhost:8082.");
});
```

Utilizando una variable podríamos crear un contador de tal modo que se concatene un nuevo número entero para cada línea adicionada al archivo de texto.

¿Se les ocurre alguna otra idea?  
¡Pruébenla ahora mismo!



# Ejercicio 4



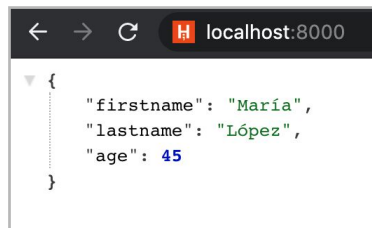


# Ejercicio 4

1. Crear una carpeta llamada `ejercicio_json`.
2. Crear dos archivos: `index.js` y `persona.js`.
3. En `index.js` crear un servidor con `createServer`.
4. En `persona.js` crear un objeto JavaScript que contenga datos de una persona. Ejemplo:  

```
{ firstname: "María", lastname: "López", age: 45 };
```
5. En `index.js` se debe importar (requerir) `persona.js`, y colocar su contenido en una variable llamada `persona`. Será necesario investigar cómo hacer esta importación.
6. En el *callback* de `createServer` se debe escribir un código tal que cuando un navegante ingrese al sitio, aparezcan los datos de la persona en formato JSON.
7. El servidor deberá estar escuchando en el puerto 8000.
8. Ingresar a `http://localhost:8000` y ver el JSON en la pantalla.

⚠ Verificar que el navegador realmente reciba una respuesta en formato JSON, y no un texto plano.





# Ejercicio 5



# Ejercicio 5

1. Crear una carpeta llamada `ejercicio_rutas`.
2. Crear un archivo: `index.js`.
3. La idea es levantar un servidor en el puerto 8000, pero la idea ahora es hacer funcionar estas 3 URLs. Para cada URL se debe mostrar un texto diferente en la pantalla.
  - a. <http://localhost:8000> → “Home”.
  - b. <http://localhost:8000/productos> → “Productos”.
  - c. <http://localhost:8000/contacto> → “Contacto”.
4. Probablemente tengan que acceder al atributo `url` del objeto `request` (ver más [aquí](#)).
5. Mejorar la solución moviendo la lógica de rutas a un archivo separado llamado `routes.js`. El código debería funcionar igual que antes, pero ahora las responsabilidades deberían quedar mejor repartidas gracias a la modularización.