

# Desarrollo Backend

Bienvenid@s

Servidores Web II

Clase 05



Pon a grabar la clase



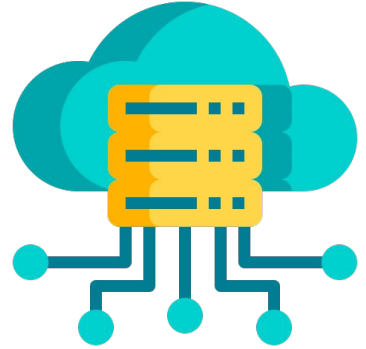
# Temario

- Usos de un servidor web
  - Crear un servidor de sitios web
  - Estructura de archivos y carpetas en Node
- Node.js como servidor de un sitio web
- El módulo path
  - Declaración
  - Métodos
- El módulo fs
  - manejo de errores
  - `__dirname`
- Creación de un servidor de sitios web
- Probar su funcionamiento
- Enviar recursos asociados al sitio web



# Usos de un servidor web

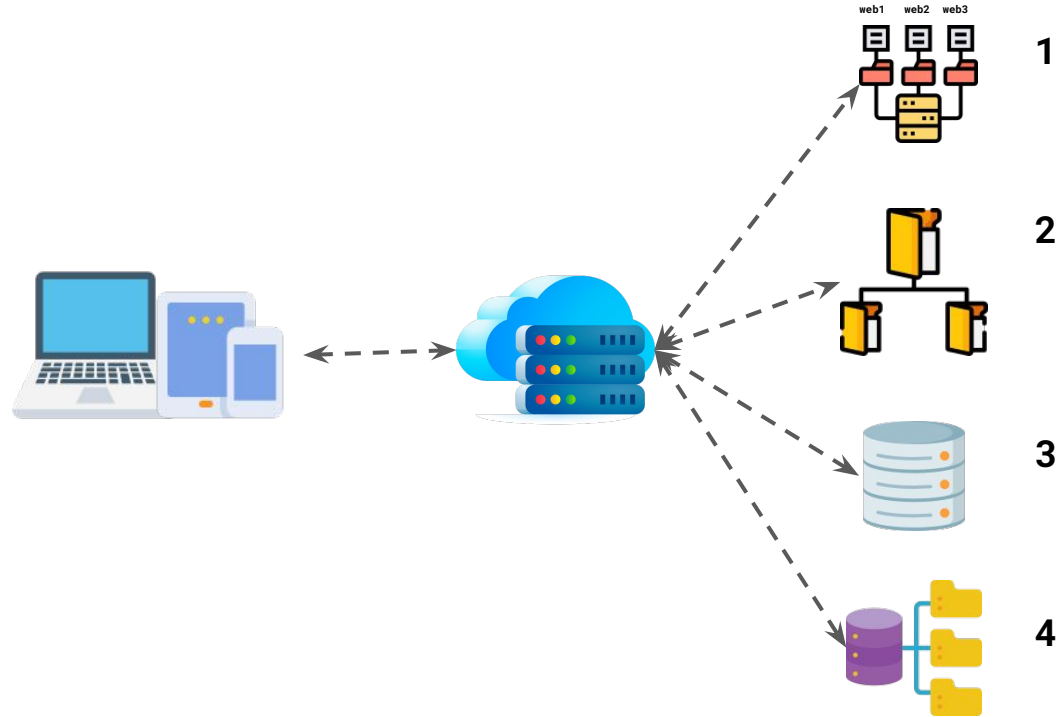
Como hemos podido ver con nuestra primera aplicación Node.js, un servidor web es una herramienta más que imprescindible hoy.



La misma facilita la interacción entre diferentes aplicaciones de software, y los datos y/o recursos que estos buscan consumir.

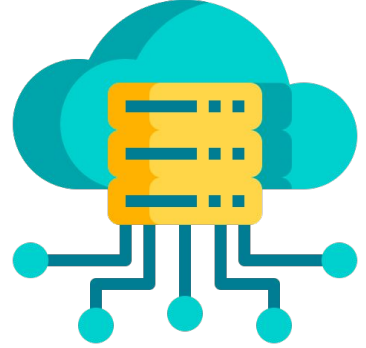
# Usos de un servidor web

Un servidor web puede proveer no solo datos estáticos provenientes de una **base de datos SQL<sup>3</sup> o NoSQL<sup>4</sup>**, recursos del tipo **archivos y carpetas<sup>2</sup>** alojados en el servidor y, por supuesto, el **acceso a uno o más sitios web<sup>1</sup>**, entre otros tantos servicios.



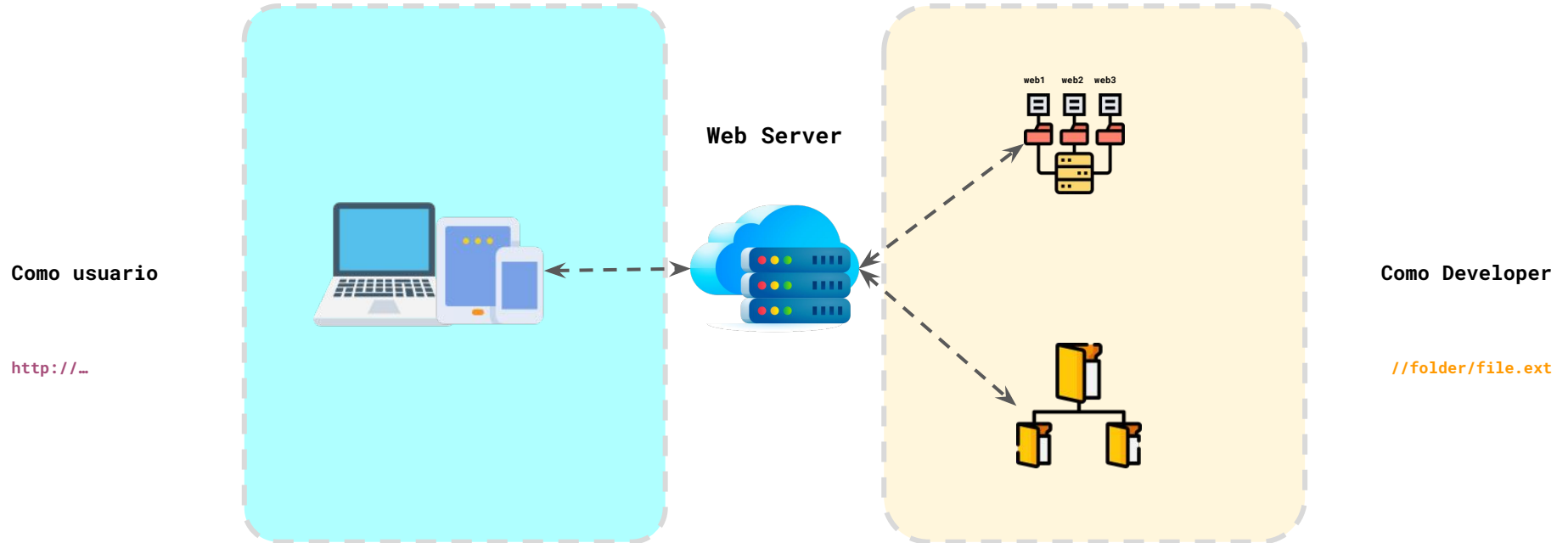
# Usos de un servidor web

Y como veíamos en el gráfico anterior, el uso de este servidor web puede aplicarse, también, a servir un sitio web convencional y recursos asociados a este.



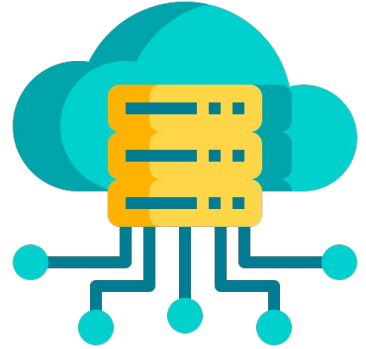
**Veamos entonces qué actores debemos tener presentes para poder desarrollar esta funcionalidad con nuestra aplicación backend.**

# Usos de un servidor web



# Usos de un servidor web

**Y para que Node.js interactúe con archivos y carpetas dentro del sistema operativo donde actualmente se ejecuta, utiliza una serie de módulos dedicados a realizar este tipo de interacciones, de una manera fácil y práctica.**



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

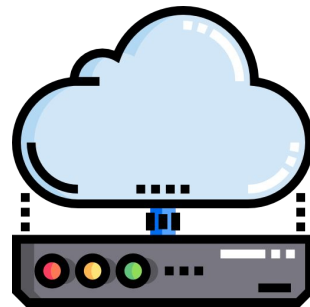


# Node.js como servidor de un sitio web

# Node.js como servidor de un sitio web

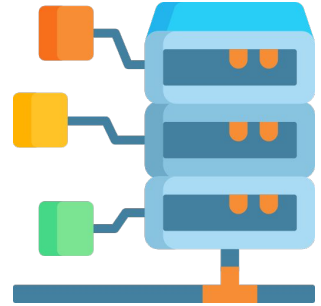
Ya interactuamos con un servidor web construido con Node.js para responder a peticiones simples de diferentes clientes.

Avancemos entonces hacia el siguiente nivel, construyendo un nuevo servidor web que provea el acceso a un sitio web frontend.



# Node.js como servidor de un sitio web

Para llevar adelante esta tarea, tenemos que pensar que nuestra aplicación web frontend, la cual debe ser servida mediante nuestro servidor web Node.js, estará alojada en una subcarpeta dentro del proyecto backend construido con Node.

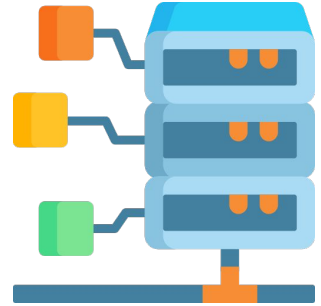


# Node.js como servidor de un sitio web

El contenido de nuestro sitio web, el cual será servido a través de la aplicación Node.js, se debe interpretar como un sitio web estático.

Serviremos el archivo HTML principal, cuando la URL del servidor web Node.js sea accedido.

Para ello, se sumarán algunos actores nuevos a nuestro desarrollo backend.



# El módulo Path

# El módulo Path

**El módulo path** de Node.js proporciona un conjunto de métodos que permiten trabajar con rutas de archivos y directorios, de forma independiente al sistema operativo en el cual se ejecuta Node.js.

En particular, este módulo permite trabajar con rutas en formato de cadena y obtener información sobre las mismas, como el nombre del directorio, el nombre del archivo, la extensión del archivo, entre otros.



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# El módulo Path

Propiedades y Métodos	Descripción
<code>path.sep</code>	Corresponde a una cadena que contiene el separador de rutas del sistema de archivos. En sistemas Unix el separador es <code>/</code> mientras que en, Windows es <code>\</code> .
<code>path.delimiter</code>	Corresponde a una cadena que contiene el separador de rutas utilizado en las variables de entorno <b>PATH</b> y <b>PATHTEXT</b> en Windows. En sistemas basados en Unix, esta propiedad es <code>:</code> . <i>(más adelante trataremos en profundidad las variables de entorno)</i>
<code>path.join(...paths)</code>	Este método recibe argumentos que representan partes de una ruta y los concatena utilizando el separador de rutas del sistema de archivos, retornando una cadena la cual representa la ruta completa.
<code>path.resolve(...paths)</code>	este método toma una serie de argumentos que representan segmentos de una ruta y devuelve la ruta absoluta resultante. Si se proporciona una ruta relativa, esta se resuelve con respecto al directorio actual.
<code>path.parse(pathString)</code>	este método toma una cadena que representa una ruta y devuelve un objeto con las partes de la ruta. El objeto tiene las siguientes propiedades: <b>root</b> , <b>dir</b> , <b>base</b> , <b>name</b> y <b>ext</b> .

Las propiedades y métodos más importantes del **módulo Path**.

Veamos, a continuación, algunos **ejemplos de código** para entender su implementación.

# El módulo Path

El **módulo Path** recibe como parámetro una o más cadenas de texto y, mediante el método **.join()**, las unifica devolviendo una estructura o ruta completa. Igual que el método de arrays, pero concatenando en el medio la barra o contrabarra, según el sistema operativo donde se ejecuta Node.js.

```
● ● ● Servidor Web

const path = require('path');

const ruta = path.join('/var', 'www', 'index.html');
console.log(ruta); // "/var/www/index.html" en sistemas basados en Unix
```



# El módulo Path

El método **.resolve()**, nos devuelve la ruta absoluta. Solo debemos aportar algunos argumentos como parte de la estructura de **archivos/carpetas**, y **.resolve()** se ocupa de obtener la estructura completa, o absoluta, según el S.O.



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

const ruta = path.resolve('www', 'index.html');
console.log(ruta); // "/home/usuario/proyecto/www/index.html" en sistemas Unix
```

# El módulo Path

```
Servidor Web

const path = require('path');

const ruta = '/home/usuario/archivo.txt';
const info = path.parse(ruta);
console.log(info);
// {
//   root: '/',
//   dir: '/home/usuario',
//   base: 'archivo.txt',
//   ext: '.txt',
//   name: 'archivo'
// }
```

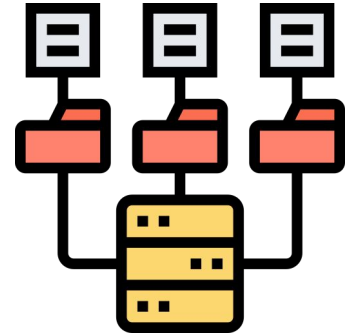
Aportando una especie de “desestructuración”, el método **.parse()** recibe la cadena de texto correspondiente a la ruta, y descompone la misma en carpetas, subcarpetas, y el archivo.

# El módulo fs

# El módulo fs

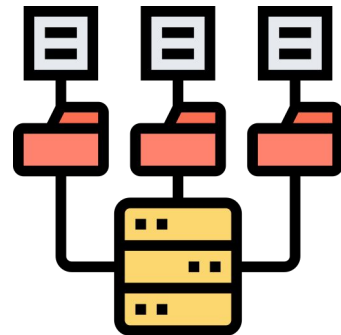
**fs** corresponde al módulo **FileSystem**.

Este es un módulo, también integrado en Node.js, el cual proporciona un conjunto efectivo de métodos para trabajar con el sistema de archivos del sistema operativo en el que se ejecuta Node.js.



# El módulo fs

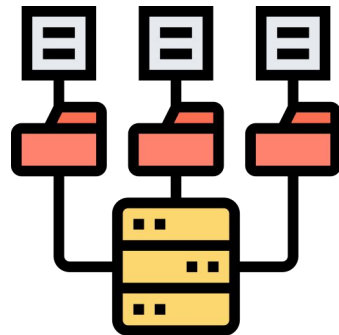
**fs** es una herramienta muy poderosa. No profundizaremos en ella en esta oportunidad, solo mencionaremos a su método **.readFile()**, que es clave cuando tenemos que leer un archivo específico, por ejemplo, documento HTML y servirlo como respuesta.



# El módulo fs

**.readFile()** toma dos argumentos:

1. el primero, corresponde a la ruta del archivo
2. el segundo, es una función de devolución que se ejecutará cuando se complete la lectura del archivo



Si ocurre algún error al leer el archivo, el primer argumento de la función de devolución de llamada contendrá un objeto de error.

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# El módulo fs

Los parámetros que recibe **.readFile()** en el siguiente ejemplo, corresponden a la ruta del archivo y al tipo de codificación del mismo (**utf8**).

```

Servidor Web

// Leer el archivo
fs.readFile(filePath, 'utf8', (err, data) => {
  if (err) {
    //Ante un error al leer el archivo, enviará una respuesta de error
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end(`Error al leer el archivo: ${err}`);
  } else {
    // Si lee el archivo correctamente, responde con el contenido del mismo
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(data);
  }
});
```

# El módulo fs

El tercer parámetro corresponde a una función que se ejecuta como respuesta, controlando un posible error o, en su defecto, enviando el archivo o documento, si este ha pasado la correspondiente validación.

```

Servidor Web

// Leer el archivo
fs.readFile(filePath, 'utf8', (err, data) => {
  if (err) {
    //Ante un error al leer el archivo, enviará una respuesta de error
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end(`Error al leer el archivo: ${err}`);
  } else {
    // Si lee el archivo correctamente, responde con el contenido del mismo
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(data);
  }
});
```

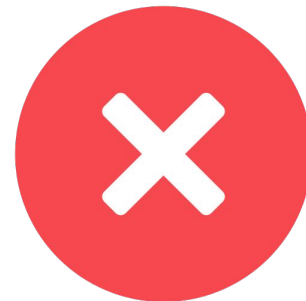


# Manejo de errores

# Manejo de errores

Los errores son otro punto importante en el manejo correspondiente al intercambio de información (**request - response**). Ante cualquier problema que ocurra, contamos con una serie de errores predefinidos bajo un estándar, que veremos en detalle más adelante.

El **error 404**, es uno de los más conocidos, desde la perspectiva de usuarios web.



# Manejo de errores

En este ejemplo podemos ver el manejo de un error, dentro del bloque de código `if()`.

Si ocurre algún tipo de error en la lógica de la aplicación, como ésta es orientada a web, aprovecharemos los **Códigos de Estado** para representar el tipo de error acontecido.

Es importante usar el código de error correcto, sobre todo si la aplicación backend es accedida por un navegador web.



```
if (err) {
  res.writeHead(500, { 'Content-Type': 'text/plain' });
  res.end(`Error al leer el archivo: ${err}`);
} else {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end(data);
}
```

# Manejo de errores



Te compartimos una publicación muy completa, y con muy pocos tecnicismos, para poder ir entendiendo en detalle lo referente a los códigos de estado HTTP:

<https://es.semrush.com/blog/codigos-de-estado-http/>

No lo profundizamos en esta etapa, para sí poder hacerlo cuando implementemos un framework JS dedicado a servidores web.

# Manejo de errores

**\_\_dirname**

**\_\_dirname** es una variable la cual nos devuelve una ruta completa de carpetas y subcarpetas, partiendo como base de la raíz del disco del sistema operativo donde Node.js está ejecutándose.

Es parte de las variables de entorno de Node.js, algo que estudiaremos con mayor detalle en los próximos encuentros.

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Manejo de errores

Tengamos presente que la estructura de carpetas que devuelve `__dirname`, varía ligeramente de acuerdo al S.O.

En Mac y Linux, utiliza un tipo de barra de separación entre carpetas, mientras que en Windows utiliza la barra opuesta, además de sumar el **prompt**, correspondiente a la unidad de disco.

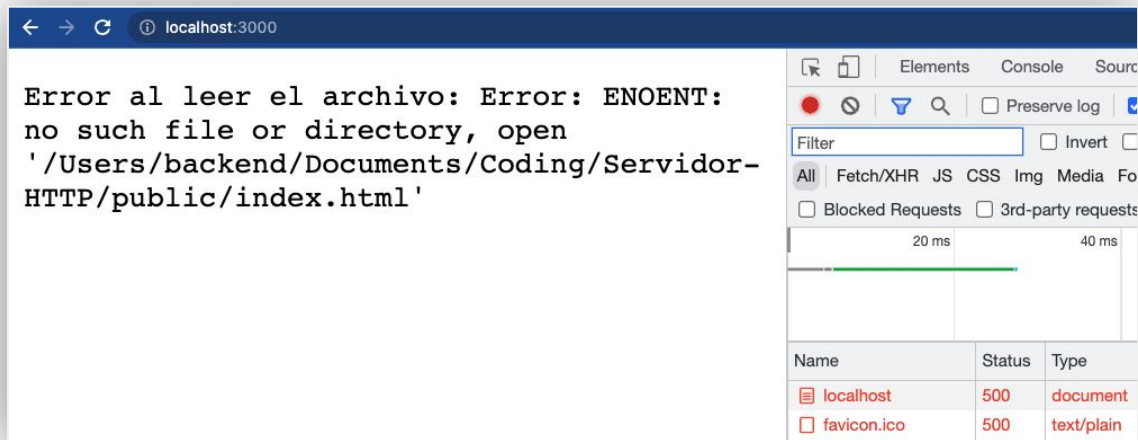
```
__dirname

console.log(__dirname);

//Ruta absoluta en sistemas Mac ó Linux
/Users/stevej/Documents/Coding/Servidor-HTTP

//Ruta absoluta en sistemas Windows
C:\Users\stevej\Documents\Coding\Servidor-HTTP
```

# Manejo de errores



Cuando el error acontece, podemos ver cómo se muestra toda la ruta de directorios y el archivo en cuestión, dentro del mensaje de error.

Este será otro tema a tratar más adelante, para que toda esta información, en algunos casos sensible, no deba ser expuesta al usuario del servicio.

# Creación de un servidor de sitios web



# Creación de un servidor de sitio web

Es hora de conjugar los temas vistos hasta aquí, en un proyecto funcional, el cual provee un sitio web a partir de la URL del servidor web.

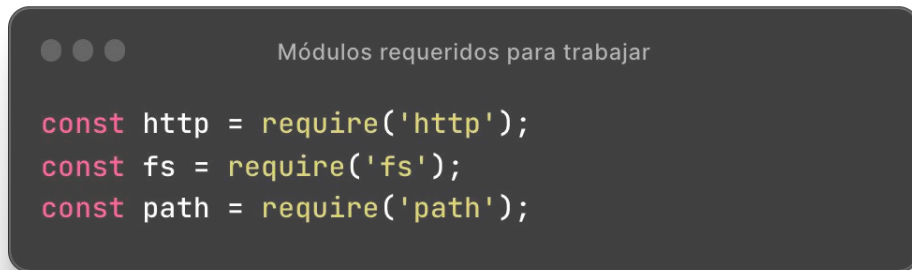
Para no perder los proyectos evolutivos, te proponemos que crees un nuevo archivo JavaScript en el proyecto Node.js de la semana anterior, definiendo su nombre como, por ejemplo: **website.js**.



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Creación de un servidor de sitio web

A dark-themed code editor window with three window control buttons (red, yellow, green) in the top-left corner. The title bar text is "Módulos requeridos para trabajar". The code inside is written in a syntax-highlighted style: "const" is pink, "http", "fs", and "path" are green, "require" is yellow, and the rest is white. The code consists of three lines: "const http = require('http');", "const fs = require('fs');", and "const path = require('path');".

```
Módulos requeridos para trabajar

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

En principio, declaramos los módulos de Node.js que utilizaremos en este proyecto:

- **HTTP**
- **File System**
- **Path**

# Creación de un servidor de sitio web



```
Crear el servidor

const server = http.createServer((req, res) => {

});
```

Creamos el servidor web HTTP, siempre definiendo un requerimiento de entrada (**req**), y la respuesta como salida (**res**).

En el interior de la **función callback**, construiremos la ruta hacia el archivo principal del sitio web.

# Creación de un servidor de sitio web

El módulo **path** y su método **join()** deben entrar en acción. En este, recordemos que definíamos la estructura de carpetas, subcarpetas, y el archivo principal a enviar mediante nuestro webserver.

Aquí, **\_\_dirname** entra en acción, como la ruta base hacia nuestro sitio web frontend.



```
const filePath = path.join(__dirname, ...);
```

# Creación de un servidor de sitio web

Crear el servidor

```
const filePath = path.join(__dirname, 'public', 'index.html');
```

Si volcamos los archivos del sitio web directamente en la carpeta **/public/**, nuestra estructura debe quedar así conformada.

Si optamos por agregar una subcarpeta contenedora del proyecto frontend dentro de **/public/**, debemos tenerla presente y agregarla como parámetro.


Crear el servidor

```
const filePath = path.join(__dirname, 'public', 'cotizador-hogar', 'index.html');
```

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Creación de un servidor de sitio web



```
Crear el servidor

fs.readFile(filePath, 'utf8', (err, data) => {
  if (err) {
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end(`Error al leer el archivo: ${err}`);
  } else {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(data);
  }
});
```

Finalmente, entra en acción el módulo File System. Este, de la mano de la instancia **fs**, invoca el método **readFile()**.

Este método recibe tres parámetros:

1. ruta del archivo
2. el formato de codificación del mismo
3. la función callback que se ejecuta

# Creación de un servidor de sitio web



```
fs.readFile(filePath, 'utf8', (err, data) => {  
  if (err) {  
    res.writeHead(500, { 'Content-Type': 'text/plain' });  
    res.end(`Error al leer el archivo: ${err}`);  
  } else {  
    res.writeHead(200, { 'Content-Type': 'text/html' });  
    res.end(data);  
  }  
});
```

La función `callBack` se ocupa, internamente, de analizar el objeto global **Error** (*recibido como primer parámetro*).

Si hay un error, debemos notificar el tipo de error. En este caso, al analizar un archivo que dice ser HTML, si el mismo no cumple la condición, se considera un error del tipo **500** - “*problema en el servidor web que hospeda el sitio web*”.

# Creación de un servidor de sitio web

```
Crear el servidor

fs.readFile(filePath, 'utf8', (err, data) => {
  if (err) {
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end(`Error al leer el archivo: ${err}`);
  } else {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(data);
  }
});
```

Si se pudo leer correctamente el archivo a servir, entonces la respuesta de estado del servidor será **200 - OK**.

Finalmente, enviamos el archivo HTML como respuesta, en este caso viaja desde el parámetro **data** que recibió la función callback.



# Creación de un servidor de sitio web

```
Manejo de estados HTTP

//Cabecera de error
res.writeHead(500, { 'Content-Type': 'text/plain' });
...

//Cabecera correcta
res.writeHead(200, { 'Content-Type': 'text/html' });
...
```

Tengamos presente siempre que, dependiendo del estado HTTP analizado, debemos definir el atributo **Content-Type** de la cabecera, como un texto plano o como un texto, en este otro caso, del tipo HTML.

Cuando enviamos contenido HTML, este no puede ser enviado como **text/plain**, porque el navegador web lo interpretará como texto plano, y lo mostrará sin renderizar.

# Creación de un servidor de sitio web

```
Manejo de estados HTTP

const http = require('http');
const fs = require('fs');
const path = require('path');

const server = http.createServer((req, res) => {
  const filePath = path.join(__dirname, 'public', 'cotizador-hogar', 'index.html');

  fs.readFile(filePath, 'utf8', (err, data) => {
    if (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end(`Error al leer el archivo: ${err}`);
    } else {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(data);
    }
  });
});

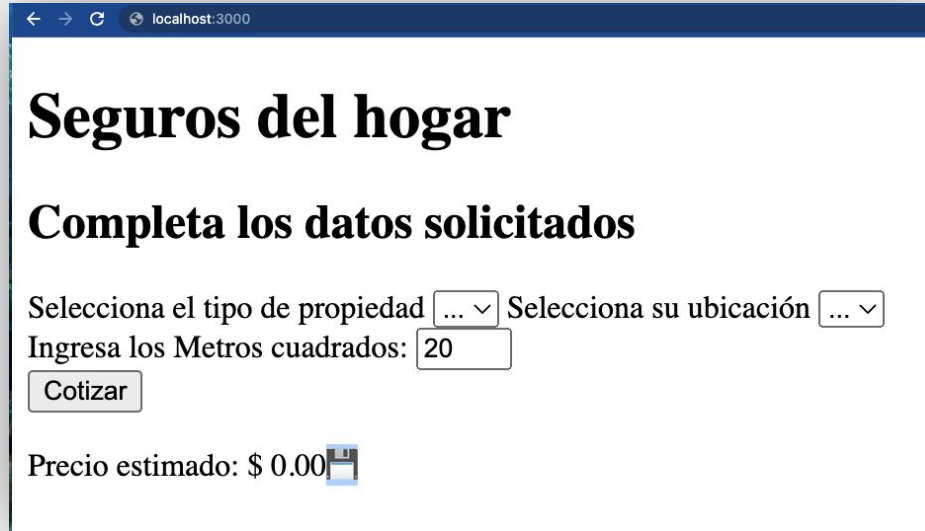
server.listen(3000, () => {
  console.log('Servidor web iniciado en el puerto 3000');
});
```

**El código completo de este ejemplo funcional.**

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Creación de un servidor de sitio web



A screenshot of a web browser window with the address bar showing 'localhost:3000'. The page has a white background and a blue header bar. The main content is a form titled 'Seguros del hogar' in a large, bold, black serif font. Below the title is a subtitle 'Completa los datos solicitados' in a smaller, bold, black serif font. The form contains two dropdown menus: 'Selecciona el tipo de propiedad' and 'Selecciona su ubicación', both with a blue arrow icon. Below these is a text input field labeled 'Ingresa los Metros cuadrados:' with the number '20' entered. To the left of the input field is a button labeled 'Cotizar' in a light blue box. At the bottom left, it says 'Precio estimado: \$ 0.00' followed by a blue floppy disk icon.

← → ↻ 🌐 localhost:3000

## Seguros del hogar

Completa los datos solicitados

Selecciona el tipo de propiedad ... ▾ Selecciona su ubicación ... ▾

Ingresa los Metros cuadrados: 20

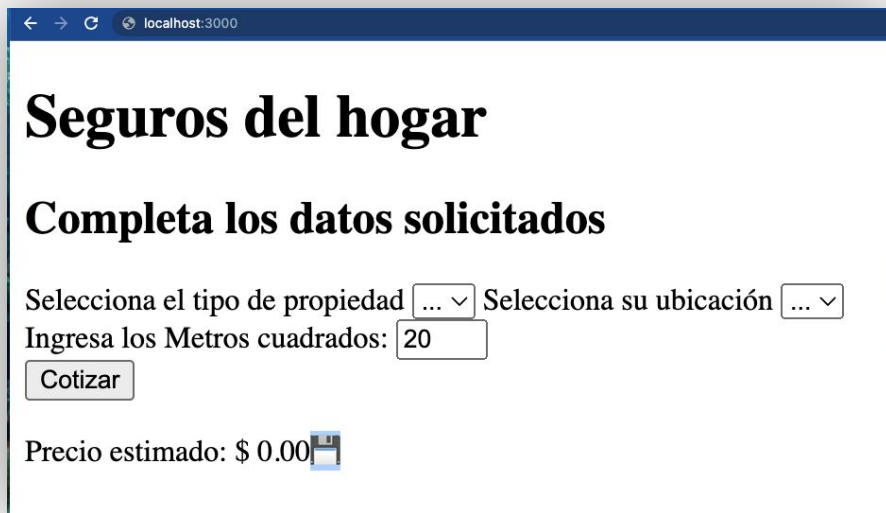
Cotizar

Precio estimado: \$ 0.00 💾

El resultado del archivo HTML  
enviado por nuestro servidor web.



# Creación de un servidor de sitio web

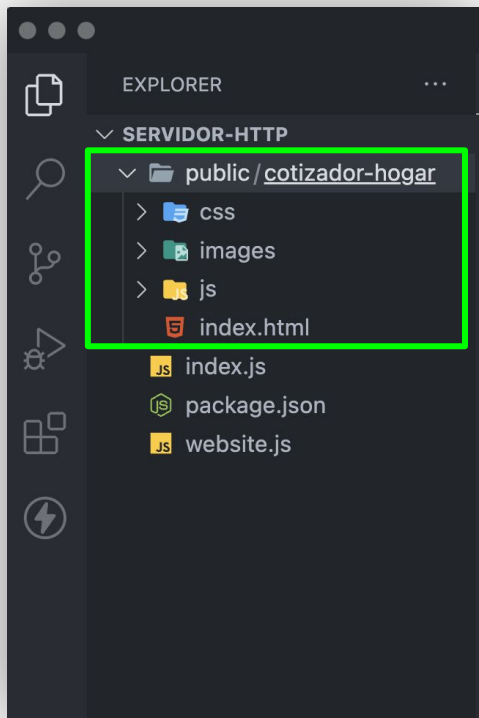


A screenshot of a web browser window displaying a form for home insurance. The browser's address bar shows 'localhost:3000'. The form has a title 'Seguros del hogar' and a subtitle 'Completa los datos solicitados'. It contains two dropdown menus for 'Selecciona el tipo de propiedad' and 'Selecciona su ubicación', a text input for 'Ingresa los Metros cuadrados' with the value '20', a 'Cotizar' button, and a 'Precio estimado: \$ 0.00' label with a small icon of a document with a checkmark.



Si usamos un HTML simple para probar el servidor web, el **resultado estará OK**. Ahora, si usamos un proyecto frontend funcional el cual incluye web fonts, CSS, frameworks CSS, archivos JS, imágenes, etcétera, seguramente no es el tipo de resultado que esperábamos.

# Creación de un servidor de sitio web



Si revisamos la estructura de nuestro proyecto frontend de ejemplo, veremos que la misma tiene subcarpetas donde se alojan los diferentes recursos vinculados en el documento HTML.

Aquí seguramente está parte de la respuesta a nuestra incógnita.



# Servir un sitio web con todos sus recursos asociados

# Creación de un servidor de sitio web

Si el documento HTML enviado al navegador contiene enlaces a recursos CSS y JavaScript, el navegador intentará descargarlos y procesarlos de forma independiente.

Si los enlaces a los recursos están definidos correctamente y el servidor tiene configurado el **MIME Type** correcto para cada recurso, el navegador debería ser capaz de descargar y procesar los recursos correctamente.

# Creación de un servidor de sitio web

```
Manejo de estados HTTP

const http = require('http');
const fs = require('fs');
const path = require('path');

const server = http.createServer((req, res) => {
  const filePath = path.join(__dirname, 'public', 'cotizador-hogar', 'index.html');

  fs.readFile(filePath, 'utf8', (err, data) => {
    if (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end(`Error al leer el archivo: ${err}`);
    } else {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(data);
    }
  });
});

server.listen(3000, () => {
  console.log('Servidor web iniciado en el puerto 3000');
});
```

En el ejemplo funcional visto hasta aquí, sólo contemplamos el **MIME Type** (*Content-Type de la respuesta HTTP*), para servir archivos HTML.

El módulo HTTP requiere ser más estricto y referenciar cada uno de los posibles tipos de archivos que viajarán como recursos, junto al documento HTML enviado.

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO



# Creación de un servidor de sitio web

```
Manejo de estados HTTP

const http = require('http');
const fs = require('fs');
const path = require('path');

const server = http.createServer((req, res) => {
  const filePath = path.join(__dirname, 'public', 'cotizador-hogar', 'index.html');

  fs.readFile(filePath, 'utf8', (err, data) => {
    if (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end(`Error al leer el archivo: ${err}`);
    } else {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(data);
    }
  });
});

server.listen(3000, () => {
  console.log('Servidor web iniciado en el puerto 3000');
});
```

Por lo tanto, para servir archivos CSS, JavaScript, etcétera, debemos agregar casos adicionales dentro de una instrucción **switch**, que establezcan el tipo MIME apropiado para cada tipo de archivo.

Veamos, a continuación, un ligero cambio en el código que debemos aplicar dentro del método **createServer()**.

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Creación de un servidor de sitio web

```
Manejo de estados HTTP

let filePath = path.join(__dirname, 'public', 'cotizador-hogar',
    req.url === '/' ? 'index.html' : req.url);

let extname = path.extname(filePath);
let contentType = 'text/html';

switch (extname) {
  case '.js':
    contentType = 'text/javascript';
    break;
  case '.css':
    contentType = 'text/css';
    break;
  case '.json':
    contentType = 'application/json';
    break;
  case '.png':
    contentType = 'image/png';
    break;
  case '.jpg':
    contentType = 'image/jpg';
    break;
  case '.wav':
    contentType = 'audio/wav';
    break;
}
```

Dentro del método **.join()** agregamos una regla del tipo **Operador Ternario**, donde evaluamos el nombre del archivo o recurso a enviar.

Luego, sumamos una estructura **switch** la cual evalúa como parámetro al archivo definido en la variable **filePath**.

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Creación de un servidor de sitio web

```
Manejo de estados HTTP

let filePath = path.join(__dirname, 'public', 'cotizador-hogar',
    req.url === '/' ? 'index.html' : req.url);

let extname = path.extname(filePath);
let contentType = 'text/html';

switch (extname) {
  case '.js':
    contentType = 'text/javascript';
    break;
  case '.css':
    contentType = 'text/css';
    break;
  case '.json':
    contentType = 'application/json';
    break;
  case '.png':
    contentType = 'image/png';
    break;
  case '.jpg':
    contentType = 'image/jpg';
    break;
  case '.wav':
    contentType = 'audio/wav';
    break;
}
```

Dependiendo del tipo de extensión del mismo, se ajusta el **Content-Type** apropiado, en este caso, en una variable independiente.

Así será más fácil aplicarlo luego al momento de enviar el contenido o recurso al cliente.

# Creación de un servidor de sitio web

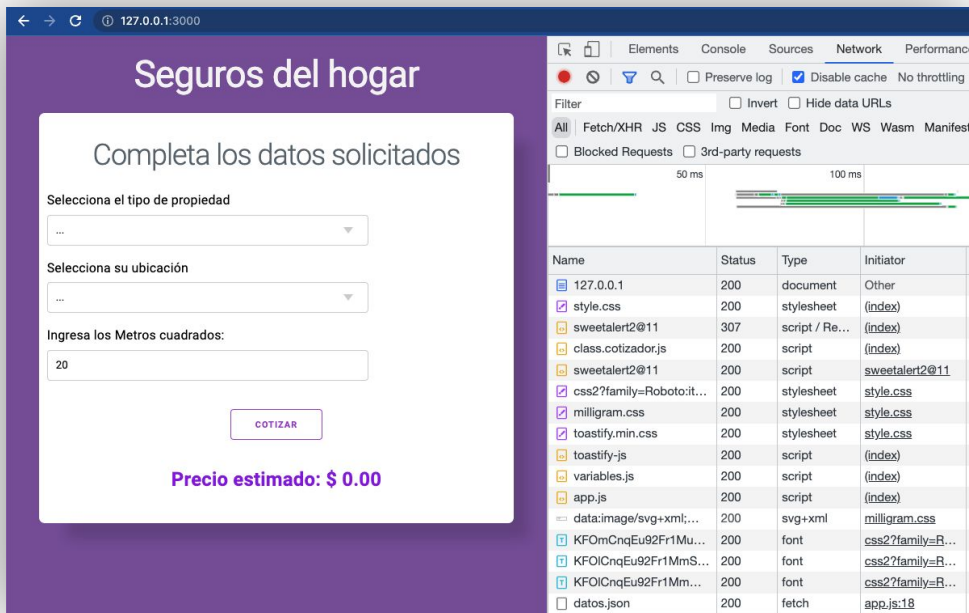
En el método **fs.readFile()** también sumamos un pequeño control de errores ante la posibilidad de identificar una falla al leer el archivo o recurso asociado, previo a enviarlo al cliente.

De esta forma, el resultado final del proyecto, deberá llegar a buen puerto y, así, podremos ver el sitio web frontend funcionando tal como se esperaba.

```
Manejo de estados HTTP

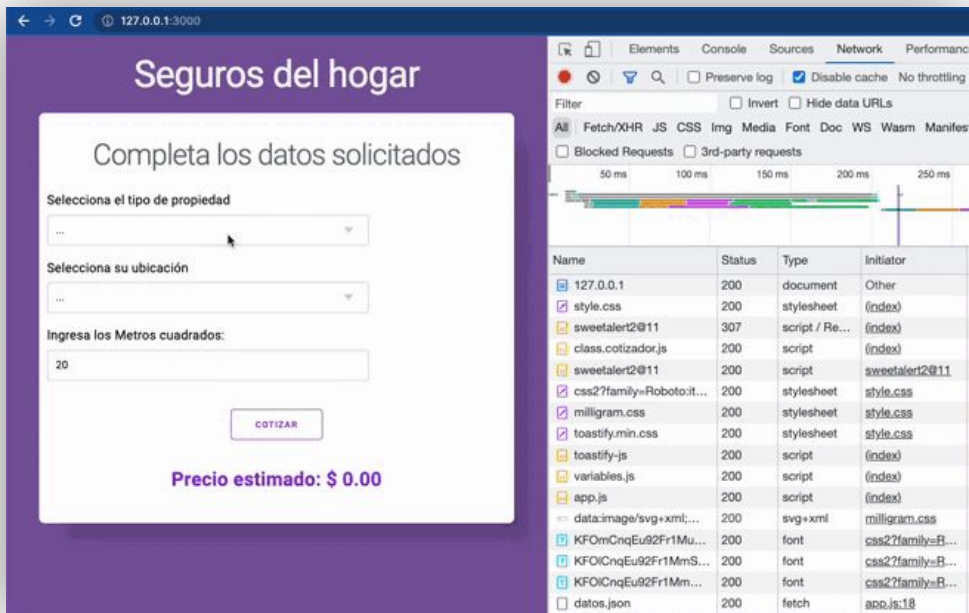
fs.readFile(filePath, (err, content) => {
  if (err) {
    if (err.code === 'ENOENT') {
      res.writeHead(404, { 'Content-Type': 'text/html' });
      res.end('<h1>404 Not Found</h1>');
    } else {
      res.writeHead(500, { 'Content-Type': 'text/html' });
      res.end(`Error interno del servidor: ${err.code}`);
    }
  } else {
    res.writeHead(200, { 'Content-Type': contentType });
    res.end(content, 'utf-8');
  }
});
```

# Creación de un servidor de sitio web



Finalmente podemos ver nuestro sitio web frontend, completamente funcional y con todos sus recursos asociados.

# Creación de un servidor de sitio web



Asegurémonos de probar en detalle, para así validar de que todos sus recursos asociados (*locales y remotos*), funcionan de acuerdo a cómo está estipulada su lógica.

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Creación de un servidor de sitio web



Estas son algunas de las particularidades que tiene el uso del módulo HTTP en proyectos de creación de servidores web.

Por suerte, nuestros próximos desarrollos de servidores serán de la mano de un Framework JS, el cual nos simplificará toda esta lógica, muchas veces innecesaria, al momento de tener que servir sitios y recursos frontend en una aplicación backend.

# Espacio de trabajo



# Espacio de trabajo

Pongamos en práctica este repaso general de la sintaxis básica y moderna de JavaScript, a través de un conjunto de ejercicios.

**Tiempo estimado:** 20 minutos. 

```
const questions = [ 'dudas', 'consultas', '🤔' ]
```



```
> node gracias.js
```