

Node y el gestor de paquetes NPM (Parte II)

Interfaces de línea de comando con Yargs

Competencias

- Reconocer los parámetros básicos del paquete Yargs para crear una interfaz de línea de comando
- Implementar una interfaz de línea de comando con el paquete Yargs para levantar una aplicación Node

Introducción

En este capítulo veremos cómo crear una interfaz de línea de comando, personalizada con uno de los paquetes más conocidos de NPM, Yargs. Esto nos permitirá construir un entorno aún más completo, en donde no solo tendremos la lógica en el servidor y la conexión con el sistema, sino que también podremos decidir cuáles serán los comandos que se deberán usar para hacer funcionar nuestra aplicación, incluso será posible agregar seguridad, al poder evaluar los valores ingresados por la línea de comandos, de esta manera por medio de credenciales se podrá autorizar al usuario y ejecutar otra aplicación.

El Paquete Yargs

Yargs es uno de los paquetes más conocidos en NPM para el desarrollo de líneas de comando, definido en su [sitio oficial](#) como “constructor de líneas de comando interactivas analizando argumentos y generando una elegante interfaz de usuario”. Es importante que sepas que una interfaz de línea de comando no es lo mismo que una interfaz gráfica, incluso sus siglas representativas no son las mismas. Las siglas “CLI” significan Command Line Interface, por otro lado tenemos las siglas “GUI” que significan Graphical User Interface.

Yargs nos ayudará a crear una CLI personalizada, pero ¿Cómo se ve una CLI? Ya has usado una y tal vez no la reconoces, déjame ayudarte, ¿Te suena el “npm install jquery”? o el ¿node index.js? Estas dos son las CLI de NPM y de Node, las puedes identificar por la primera palabra que escribes. En nuestros ejercicios estaremos ocupando de igual manera la invocación con Node de un archivo JavaScript pero posterior a este escribiremos los nuevos comandos personalizados.

Instalación

Para instalar este paquete deberás usar el siguiente comando:

```
npm install yargs
```

Una vez instalado solo necesitarás importarlo en una constante y empezar a usar su API, la cual es bastante extensa y podrás conocerla en profundidad en su [documentación oficial](#). En esta lectura iremos directamente a los métodos que nos permitirán crear una CLI de forma básica pero igual funcional.

Mi primer CLI

Dentro del objeto importado del paquete “Yargs” contamos con el método “command”, el cual, tiene el objetivo de definir los parámetros y la configuración de nuestra interfaz de línea de comando, y se usa por medio de la siguiente sintaxis:

```
.command( <comando> , <descripción>, <constructor>, <callback> )
```

A continuación te muestro una lista con la definición de cada parámetro:

- **Comando:** Acá deberás escribir un dato tipo String que representará el comando principal de nuestra CLI.
- **Descripción:** Es la descripción de nuestro comando. Cada comando podrá ser accedido próximamente en una lista, acompañados de sus descripciones para un mejor entendimiento con el usuario.
- **Constructor:** En este parámetro definiremos en forma de objeto el único o los diferentes argumentos que queremos disponer en nuestro CLI, además de sus especificaciones personales (descripción, requerido, alias). Los argumentos en Yargs se deberán escribir con 2 guiones antes, por ejemplo --argumento, o con un solo guión utilizando su alias como si se tratara de un flag (bandera), ejemplo: -a
- **Callback:** La función se ejecutará una vez que se ejecute la línea de comando. Esta función tiene una particularidad, la cual es recibir como parámetro el objeto "argv" que contendrá los argumentos escritos en la línea de comando.

Ejercicio guiado: Aplicando Yargs

Construir una aplicación que use el paquete Yargs para la definición de una interfaz de línea de comandos. El objetivo será definir un comando "saludo" que reciba como argumento tu nombre y responda con un mensaje en consola saludando y deseando un excelente día, sigue los siguientes pasos:

- **Paso 1:** Importar en una constante el paquete Yargs.
- **Paso 2:** Inicializar el método "command" para el paso de parámetros.
- **Paso 3:** Definir del comando con el primer parámetro el cual será "saludo".
- **Paso 4:** Definir la descripción del comando "saludo" como segundo parámetro del método "command".
- **Paso 5:** Definir el objeto para la configuración del constructor del comando.
- **Paso 6:** Declarar que se esperará recibir un argumento llamado "nombre".
- **Paso 7:** Definir la descripción de este argumento.
- **Paso 8:** Declarar que este argumento es requerido con un true en la propiedad "demand".

- **Paso 9:** Declarar el alias del argumento nombre, el cual será "n". Esto sirve para simplificar la declaración de un argumento recortando su mención a solo 1 letra o siglas.
- **Paso 10:** Crear la función callback la cual recibe como parámetro el objeto args que contendrá los argumentos como propiedades. A su vez la función mandará un mensaje por consola saludando con el nombre recibido como argumento.
- **Paso 11:** Concatenar el método command con el método "help" y la propiedad argv.

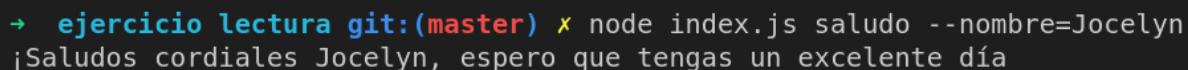
El código entonces te quedará como te muestro a continuación:

```
// Paso 1
const yargs = require('yargs')
// Paso 2
const argv = yargs
  .command(
    // Paso 3
    'saludo',
    // Paso 4
    'Comando para saludar',
    // Paso 5
    {
      // Paso 6
      nombre: {
        // Paso 7
        describe: 'Argumento para definir tu nombre',
        // Paso 8
        demand: true,
        // Paso 9
        alias: 'n',
      },
    },
    // Paso 10
    (args) => {
      console.log(`¡Saludos cordiales ${args.nombre}, espero que tengas un excelente día`)
    }
  )
// Paso 11
.help().argv
```

- **Paso 12:** Ahora probemos, abre la terminal y ejecuta el siguiente comando:

```
node index.js saludo --nombre=Jocelyn
```

Y deberás recibir por pantalla el mensaje que te muestro en la siguiente imagen:



```
→ ejercicio lectura git:(master) x node index.js saludo --nombre=Jocelyn
¡Saludos cordiales Jocelyn, espero que tengas un excelente día
```

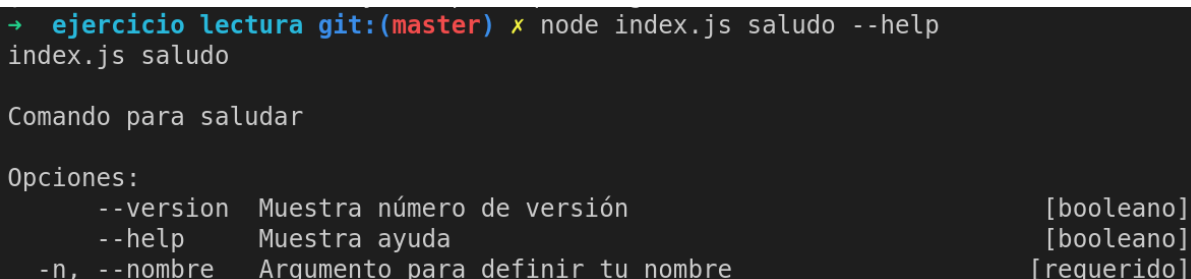
Imagen 1. Mensaje por consola luego de ejecutar una línea de comando personalizada.

Fuente: Desafío Latam.

¡Muy bien, funciona!, agreguemos el método “help” al final, esto nos ayudará a poder ocupar el comando `--help` que nos devolverá una lista con las descripciones declaradas en la configuración. Escribe el siguiente comando para visualizarlo.

```
node index.js saludo --help
```

Y deberás recibir lo que te muestro en la siguiente imagen:



```
→ ejercicio lectura git:(master) x node index.js saludo --help
index.js saludo

Comando para saludar

Opciones:
  --version  Muestra número de versión           [booleano]
  --help     Muestra ayuda                         [booleano]
  -n, --nombre Argumento para definir tu nombre [requerido]
```

Imagen 2. Mensaje por consola resultado del comando `--help`.

Fuente: Desafío Latam.

Acá como puedes ver, obtenemos el detalle de opciones posibles de ejecutar así como también las descripciones definidas.

Ejercicio propuesto (1)

Crear una interfaz de línea de comando con el paquete Yargs que defina un comando “PING” y un argumento número, devuelva el mensaje “PONG” y el número recibido concatenadamente.

Evaluando los argumentos

Ahora que sabemos que podemos usar los argumentos recibidos por línea de comandos, podríamos escribir un condicional dentro de la función callback, que evalúe si el valor recibido es igual al valor que esperamos para realizar alguna acción. Por ejemplo: “Tenemos una aplicación en un archivo aparte que se ejecutará sólo si los argumentos ingresados por la línea de comando son las credenciales de un usuario Administrador”.

¿Cómo desarrollaremos esta validación? Simple, alojamos en nuestro código un usuario y contraseña específico y evaluamos si los valores recibidos coinciden con estos, entonces de ser así la función callback podría ejecutar con el módulo “child_process” un archivo externo y diferente a la aplicación con la que estamos interactuando, es decir, que podemos ejecutar o levantar una aplicación externa a partir de la evaluación de argumentos pasados en la línea de comandos.

¿Un poco extraño cierto? Es normal, pues estamos acostumbrados a trabajar bajo un mismo archivo JavaScript, sin embargo, en el mundo real las aplicaciones están distribuidas en pequeñas miniaplicaciones, que en algunos casos funcionan como microservicios que se interconectan y se comunican para procesar una petición.

Ejercicio guiado: Evaluando los argumentos

Desarrollar una aplicación que reciba argumentos por la línea de comandos y los evalúe, estos argumentos corresponden a unas credenciales, en caso de éxito, es decir que las credenciales son las correctas, se ejecutará otra aplicación que devolverá un mensaje por consola indicando “Bienvenido al Área 51”.

Lo primero será crear un archivo nuevo llamado acceso.js con el siguiente código.

```
console.log('Bienvenido al Área 51')
```

Este archivo entonces será llamado desde el index.js solo si la condicional lo permite. Sigue los siguientes pasos para la solución de este ejercicio.

- **Paso 1:** Importar en una constante el paquete child_process.
- **Paso 2:** Definir las credenciales de acceso.
- **Paso 3:** Definir el comando “acceso”.
- **Paso 4:** Definir la descripción del comando “acceso”.

- **Paso 5:** Definir qué se requerirá el argumento "user" con sus características.
- **Paso 6:** Definir qué se requerirá el argumento "pass" con sus características.
- **Paso 7:** Declarar el operador ternario que condicione si los valores ingresados en los argumentos coinciden con las credenciales.
- **Paso 8:** Ejecutar el archivo acceso.js con el método exec del módulo child_process e imprimir por consola su respuesta.
- **Paso 9:** Devolver en el "else" del operador ternario un mensaje por consola que diga "Credenciales incorrectas"

```
const yargs = require('yargs')
// Paso 1
const child = require('child_process')
// Paso 2
const user = 'Ovni22'
const pass = 123457
const argv = yargs
  .command(
    // Paso 3
    'acceso',
    // Paso 4
    'Comando para acceder al Área 51',
    {
      // Paso 5
      user: {
        describe: 'Usuario',
        demand: true,
        alias: 'u',
      },
      // Paso 6
      pass: {
        describe: 'Contraseña',
        demand: true,
        alias: 'p',
      },
    },
    (args) => {
      // Paso 7
      args.user == user && args.pass == pass
        ? // Paso 8
          child.exec('node acceso.js', (err, stdout) => {
```

```
    err ? console.log(err) : console.log(stdout)
  })
  : // Paso 9
    console.log('Credenciales incorrectas')
  }
)
.help().argv
```

- **Paso 10:** Ejecutar el archivo index.js con el siguiente comando:

```
node index.js acceso -u=0vni22 -p=123457
```

Obtendrás lo que te muestro en la siguiente imagen:

```
→ ejercicio lectura git:(master) x node index.js acceso -u=0vni22 -p=123457
Bienvenido al Área 51
```

Imagen 3. Mensaje por consola resultado del comando --help.

Fuente: Desafío Latam

Sensacional! Ya has aprendido a crear una interfaz de línea de comando con el paquete Yargs, tus conocimientos han crecido y ahora eres un mejor desarrollador, sin embargo, el aprendizaje nunca acaba por lo que eventualmente conocerás nuevas herramientas, tecnologías, librerías, frameworks y paquetes.

Y hablando de paquetes, prepárate porque ahora aprenderás a usar uno que estoy seguro te gustará.

Ejercicio propuesto (2)

Crear una interfaz de línea de comando con el paquete Yargs que defina un comando "adulto" y un argumento "edad", que al ser ejecutado evalúe si el valor del argumento es mayor a 18, de ser así devolver el mensaje "Mayor de edad", de lo contrario devolver "Menor de edad"

Procesamiento de imágenes con Jimp

Competencia

- Construir un servidor que al ser consultado devuelva una imagen procesada con el paquete Jimp

Introducción

En este capítulo aprenderás a utilizar un paquete de NPM que trabaja con imágenes y las procesa para realizar efectos y cambios particulares.

El paquete Jimp te brindará la posibilidad de programar nuevas funcionalidades en tus aplicaciones un poco diferente de los procesos administrativos tradicionales, en esta ocasión tendrás un primer encuentro con el procesamiento de imágenes en donde aprenderás cómo redimensionar y aplicar filtros a una imagen de internet.

El Paquete Jimp

En el desarrollo hay espacio para toda innovación, tenemos paquetes que ayudan a la administración, gestión, consulta de recursos externos, ejecución de aplicaciones ajenas, a crear servidores, etc. Sin embargo, no hemos hablado de las herramientas “artísticas” que aplican mayormente a los requerimientos funcionales para el desarrollo de aplicaciones de entretenimiento.

El paquete Jimp pertenece a los paquetes que trabajan con archivos multimedia, en este caso con imágenes específicamente, ofreciéndonos diferentes funcionalidades como por ejemplo, el redimensionamiento, recorte, filtros, orientación, brillo, contraste, entre otros, que puedes conseguir en su [repositorio](#) de NPM.

Ejercicio guiado: Aplicando el paquete Jimp

Crear un servidor que al consultarse devuelva una imagen redimensionada a 250 pixeles de ancho y un alto automático, además de aplicarle el filtro sepia. ¿Y cómo se hace eso? A continuación te muestro la sintaxis correspondiente a la instancia de Jimp que usarás para lograrlo:

```
Jimp.read(<dirección de la imagen>, <callback(err,imagen)>)
```

Dentro del callback recibiremos como primer parámetro el error en caso de existir y como segundo parámetro la instancia de la imagen leída. Lo siguiente será usar la sintaxis con la instancia de la imagen.

```
imagen
.resize(<ancho>, <alto>)
.sepia()
.writeAsync(<nombre_nueva_imagen>)
.then()
```

La dirección de la imagen puede apuntar a una imagen local o remota, en mi caso usaré la URL de una imagen de google, siéntete libre de usar la que quieras para el experimento.

Ahora que sabemos cómo usar el paquete Jimp, prosigue con los siguientes pasos para la solución de este ejercicio.

- **Paso 1:** Importar en una constante el paquete Jimp.
- **Paso 2:** Importar en una constante el módulo http.

- **Paso 3:** Importar en una constante el módulo fs.
- **Paso 4:** Crear un servidor con el método de http createServer.
- **Paso 5:** Usar el método read del objeto Jimp definiendo como primer parámetro la url de una imagen.
- **Paso 6:** Aplicar los siguientes métodos, siguiendo la secuencia:
 - Método resize definiendo como primer parámetro 250 y como segundo parámetro el método AUTO del objeto Jimp para el cálculo automático del height.
 - Método sepia para aplicar el filtro sepia a la imagen.
 - Método writeAsync declarando el nombre del archivo procesado a almacenar.
- **Paso 7:** Usar el módulo fs para la lectura del archivo creado.
- **Paso 8:** Definir la cabecera de la respuesta a la consulta del servidor con un Content-Type en valor image/jpeg y terminar la consulta devolviendo la data del archivo leído con el readFile.

```
// Paso 1
const Jimp = require('jimp')
// Paso 2
const http = require('http')
// Paso 3
const fs = require('fs')
// Paso 4
http
  .createServer((req, res) => {
    // Paso 5
    Jimp.read('https://miviaje.com/wp-content/uploads/2016/05/shutterstock_37174700.jpg', (err, imagen) => {
      // Paso 6
      imagen
        .resize(250, Jimp.AUTO)
        .sepia()
        .writeAsync('img.png')
        .then(() => {
          // Paso 7
          fs.readFile('img.png', (err, Imagen) => {
            // Paso 8
            res.writeHead(200, { 'Content-Type': 'image/jpeg' })
            res.end(Imagen)
```

```
    })  
  })  
})  
.listen(3000, () => console.log('Server on'))
```

Como referencia la próxima imagen que verás es la imagen original que usé para este ejercicio.

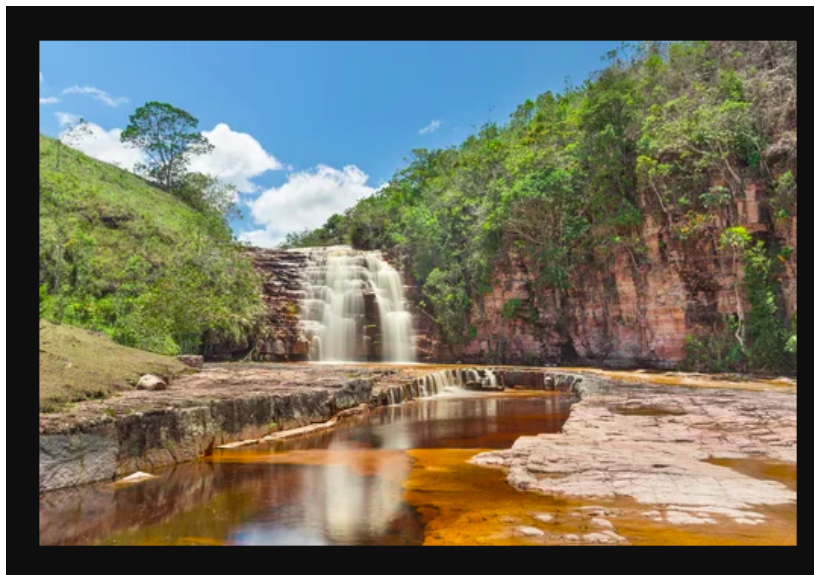


Imagen 4. Imagen original sin procesar.
Fuente: Desafío Latam

Y ahora si levantas el servidor creado y la consultas desde el navegador deberás recibir la imagen procesada, tal y como te muestro en la siguiente imagen.



Imagen 5. Imagen procesada.
Fuente: Desafío Latam

¡Felicidades! oficialmente has tenido tu primer encuentro con algoritmos de procesamiento de imágenes. Como dato extra, esto es 100% escalable y si lo aplicamos en un plano más real, la dirección de la imagen debería ser asignada por un usuario en un formulario desde el cliente.

Ejercicio propuesto (3)

Crear un servidor que al ser consultado devuelva una imagen procesada con los métodos `resize`, `grayscale` y `quality`.

El método `grayscale` al igual que el `sepia` no necesitan recibir un parámetro, no obstante el `quality` recibe un número del 0 al 100 indicando el porcentaje de calidad. Por ejemplo, si quisieras bajar a un 60% la calidad de una imagen ocuparían el método `quality` como te muestro a continuación

```
quality(60)
```

Bajando aplicaciones

Competencia

- Ejecutar procedimiento para bajar una aplicación Node utilizando la línea de comandos

Introducción

Todo proceso ejecutado o en ejecución en los sistemas operativos es representado por un identificador numérico y una aplicación de Node no es la excepción.

En este capítulo aprenderás cómo bajar o cancelar tus aplicaciones a través de la línea de comando, gracias a una combinación de comandos que ejecute la finalización de este proceso a través de su PID. Usaremos un servidor básico para mantener activo un proceso y en una terminal diferente se ejecutarán los comandos correspondientes para bajar al servidor. Conocer sobre procesos y sus identificadores te darán un valor agregado como desarrollador full stack developer.

Cancelando una aplicación en proceso

Seguramente te has dado cuenta que cuando ejecutas un servidor la terminal queda en un estado diferente a cuando ejecutamos una simple aplicación que devuelve un mensaje por consola. Esto sucede, porque para que el servidor esté “encendido” se necesita que el proceso ejecutado se mantenga activo.

Todos los procesos en los sistemas operativos son identificados por un número PID (Identificador de Proceso) y conociendo este identificador podríamos voluntariamente darlo de baja o en otras palabras cancelarlo, por consecuencia liberar el puerto que esté ocupando.

El número PID se puede conseguir como propiedad dentro del objeto “process”. Este valor no es fijo, por lo que irá variando por cada nuevo levantamiento ejecutado.

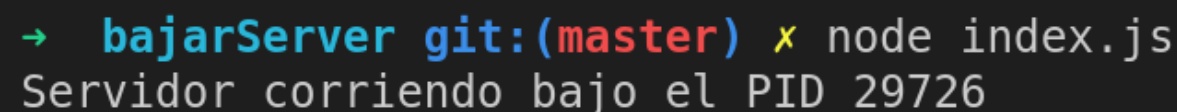
Para entenderlo mejor, será necesario tener un servidor, sin embargo, no necesita tener ningún proceso internamente, simplemente ocupar un puerto. Con el siguiente código como bien sabes bastará para lograr esto.

```
const http = require('http')
http.createServer((req, res) => {

})
.listen(3000,
() => console.log('Servidor corriendo bajo el PID', process.pid))
```

Nota que en el mensaje que se está enviando por consola está concatenado el PID del proceso, con el objetivo de obtener este número al levantar el servidor.

Si procedes a levantar el servidor deberás recibir un mensaje como el que te muestro en la siguiente imagen.



```
→ bajarServer git:(master) x node index.js
Servidor corriendo bajo el PID 29726
```

Imagen 6. Mensaje por consola indicando el PID del proceso.
Fuente: Desafío Latam

Como puedes notar recibí el número “29726”, el cual representa el identificador del proceso que mi sistema operativo ejecutó para levantar el servidor. Para cancelar este proceso de forma manual bastará con presionar Ctrl + C, sin embargo, lo queremos hacer por medio de

la línea de comando. Esto se consigue diferente según el sistema operativo, en MAC y LINUX disponemos del siguiente comando:

```
kill -9 <pid>
```

Pero en windows se usará el siguiente comando.

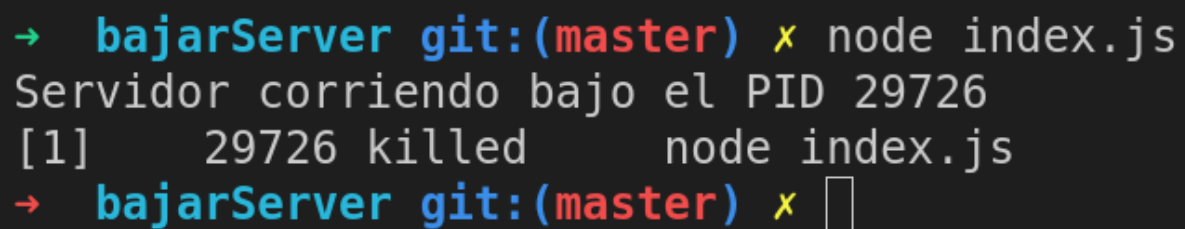
```
taskkill /F /PID <pid>
```

En donde deberás cambiar <pid> por el número identificador del proceso.

Ahora intentemos cancelar el servidor levantado, para esto deberás abrir otra terminal y proceder con la instrucción, en mi caso ocuparé el siguiente comando:

```
kill -9 29726
```

Obteniendo como resultado lo que te muestro en la siguiente imagen



```
→ bajarServer git:(master) x node index.js
Servidor corriendo bajo el PID 29726
[1] 29726 killed node index.js
→ bajarServer git:(master) x
```

Imagen 7. Mensaje por consola indicando que el proceso fue cancelado.

Fuente: Desafío Latam

Como podrás apreciar, el proceso fue cancelado y como consecuencia el servidor fue bajado.

Devolviendo sitios web estáticos

Competencias

- Reconocer la importancia de alojar archivos en un servidor a través de una ruta para su importación en un HTML devuelto por el servidor
- Construir y levantar un servidor con Node y el paquete nodemon para servir sitios web estáticos a partir de una consulta HTTP

Introducción

No es un secreto para nadie que el desarrollo web es un complemento entre el desarrollo frontend y backend, sin embargo, estos dos mundos están altamente relacionados hasta el punto en el que uno puede ser la consecuencia de otro, es decir, que gracias al backend podríamos ocupar el frontend. ¿Cómo es posible esto? Resulta, que el poder que disponemos en el lado del servidor tiene varias aplicaciones, no solo funciona para servir datos, sino que también puede servir sitios web completos.

En este capítulo, veremos que para devolver un sitio web desde el servidor necesitamos crear una ruta especialmente para código HTML y otra ruta para el código CSS, puesto que cada documento debe ser devuelto con una cabecera que indica el formato de su contenido. Finalmente el HTML consultará la ruta del CSS creada en el servidor para importar los estilos.

Entendiendo y aplicando esta arquitectura de desarrollo podrás crear sitios web que estén finalmente alojados y devueltos desde tu servidor, todo enlazado en un mismo sistema logrando compartir sus herramientas y funciones.

Devolviendo HTML

Para devolver HTML puro desde el servidor tenemos 2 formas, devolverlo en formato String o devolviendo la data de un documento leído entre los archivos del servidor. Ambas formas deberán tener una instrucción que especifique en la cabecera de la respuesta, que el contenido devuelto deberá ser interpretado como HTML, la instrucción es la que te muestro en el siguiente código:

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

Esto se deberá situar siempre que se quiera devolver contenido HTML

Formato String

Usa el siguiente código para la creación de un servidor básico disponible a través del puerto 3000 que al ser consultado devolverá un HTML dentro de un string:

```
const http = require('http')
http
  .createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html' })
    res.end(`
      <p> Desafio <b>LATAM</b> </p>
    `)
  })
  .listen(3000, () => console.log('Servidor encendido'))
```

Ahora procede a levantar el servidor con nodemon, porque estaremos haciendo algunos cambios a lo largo del capítulo.

Ejecuta el siguiente comando para iniciar un proyecto NPM.

```
npm init -y
```

Se incluye el flag -y para aceptar todas las preguntas por defecto y obtener inmediatamente el package.json.

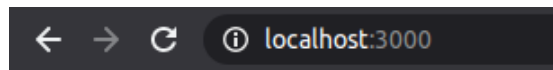
Ahora instala nodemon con el siguiente comando:

```
npm install nodemon
```

Posteriormente levanta el servidor con el siguiente comando:

```
nodemon index.js
```

Ahora consulta el servidor desde el navegador y obtendrás lo que te muestro en la siguiente imagen:



Desafío **LATAM**

Imagen 8. Imagen procesada.
Fuente: Desafío Latam

Se concluye que el navegador está interpretando código HTML porque la palabra “LATAM” tiene aplicado el estilo de la etiqueta bold.

Ejercicio propuesto (4)

Crear un servidor que al ser consultado devuelva una una etiqueta img con un ancho de 600 pixeles formato String.

Data de un archivo HTML

La forma ideal de devolver HTML debe ser leyendo el contenido de un fichero de extensión HTML y devolviendo este contenido al consultarse el servidor.

Para ver un ejemplo crea un documento HTML llamado index.html con el siguiente código.

```
<div>ADL</div>
<style>
  div {
    position: absolute;
    border-radius: 50%;
    padding: 10px;
    background: darkgreen;
    color: white;
  }
</style>
```

Como puedes ver se está usando las etiquetas div y style. Ahora en el siguiente código te muestro como usando el módulo File System en el servidor, se consulta la data de este documento creado y devolviendolo como respuesta.

```
const http = require('http')
const fs = require('fs')
http
  .createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html' })
    fs.readFile('index.html', 'utf8', (err, data) => {
      res.end(data)
    })
  })
  .listen(3000, () => console.log('Servidor encendido'))
```

Ahora si consultas al servidor desde el navegador obtendrás lo que te muestro en la siguiente imagen.

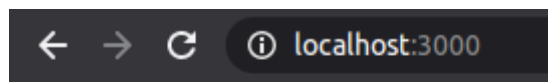


Imagen 9. Data de un documento HTML obtenido en el navegador desde el servidor.
Fuente: Desafío Latam

Excelente, con esto estamos devolviendo un documento HTML que podría ser todo un sitio web, pero espera, un sitio web completo no se debe escribir en un mismo documento HTML sino que debe estar distribuido en archivos HTML, CSS y JavaScript. Acá nos topamos con un detalle importante que es la importación de archivos desde el HTML.

Ejercicio propuesto (5)

Crear un servidor que al ser consultado devuelva el siguiente código HTML

```
<script>
  alert('Soy el archivo Alerta.html ¡Funciona!')
</script>
```

Importando documentos desde rutas específicas en el servidor

Como bien sabes en HTML se ocupa la etiqueta head para importar las dependencias o por último en caso de JavaScript usar el final de la etiqueta body. Si intentamos hacer esto con lo que tenemos hasta ahora tendremos un problema, y es que sea cual sea la ruta que decidamos consultar para obtener los estilos o los scripts, si esta no existe como ruta en el servidor, no obtendremos nada.

En el frontend no teníamos este problema porque siempre tuvimos definido en la URL la ruta del sistema operativo necesaria para llegar hasta los documentos, partiendo desde el disco duro. En esta ocasión, todo lo que le consultemos al servidor debe salir del servidor y si necesitamos importar algún archivo, su contenido debe ser devuelto de una ruta declarada en el mismo.

¿Ok, y ahora qué? Lo primero será dividir el HTML y el CSS en archivos diferentes, luego tendremos que crear una ruta para el documento CSS. Entonces, crea un documento CSS llamado estilos.css y pega el código CSS que escribimos inicialmente en el documento HTML extrayéndose de la etiqueta style. El objetivo será conseguir el mismo resultado que pudiste ver en la imagen 9 pero en esta ocasión teniendo ambos formatos dividido en 2

archivos diferentes comunicándose por medio de rutas disponibilizadas por nuestro servidor.

El código en el documento index.html quedaría de la siguiente manera.

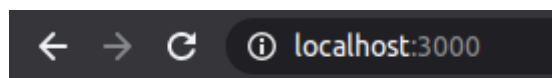
```
<head>
  <link rel="stylesheet" href="http://localhost:3000/estilos" />
</head>
<div>ADL</div>
```

Nota que estoy usando la etiqueta link dentro de la etiqueta head para importar el código CSS, además de definir que la ruta en la que buscará este código ahora incluye la url base de nuestro servidor local en el puerto 3000.

El código en el documento estilos.css quedaría de la siguiente manera.

```
div {
  position: absolute;
  border-radius: 50%;
  padding: 10px;
  background: darkgreen;
  color: white;
}
```

Bien, ya tenemos separados los códigos HTML y CSS en sus respectivos archivos. Pero, si intento ingresar nuevamente al servidor me topo con la sorpresa que se muestra en la siguiente imagen.



ADL

Imagen 10. Data de un documento HTML obtenido en el navegador desde el servidor.

Fuente: Desafío Latam

¿Qué sucedió aquí? El HTML no puede encontrar el código CSS porque la ruta especificada en la etiqueta link no está devolviendo nada. ¿Y esto por qué? Porque no hay una ruta aún creada en el servidor. He aquí la importancia de tener una ruta para cada documento que deseemos utilizar dentro del sitio web que ofreceremos en nuestro servidor.

Entonces, ahora solo falta crear las rutas correspondientes a cada archivo en el servidor. Para esto sigue los siguientes dos simples pasos:

- **Paso 1:** Crear la ruta raíz, la cual además de definir la cabecera correspondiente a código HTML devolverá la data del archivo index.html obtenido con el método readFile del módulo File System.
- **Paso 2:** Crear la ruta /estilos.css, la cual además de definir la cabecera correspondiente a código CSS devolverá la data del archivo estilos.css obtenido con el método readFile del módulo File System

```
const http = require('http')
const fs = require('fs')
http
  .createServer((req, res) => {
    // Paso 1
    if (req.url == '/') {
      res.writeHead(200, { 'Content-Type': 'text/html' })
      fs.readFile('index.html', 'utf8', (err, html) => {
        res.end(html)
      })
    }
    // Paso 2
    if (req.url == '/estilos') {
      res.writeHead(200, { 'Content-Type': 'text/css' })
      fs.readFile('estilos.css', (err, css) => {
        res.end(css)
      })
    }
  })
  .listen(3000, () => console.log('Servidor encendido'))
```

Ahora si consultas al servidor desde el navegador obtendrás lo que te muestro en la siguiente imagen.

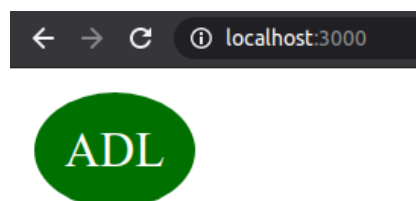


Imagen 11. Sitio web estático recibido por el servidor.
Fuente: Desafío Latam

¡Excelente! Volvimos a recibir nuestro sitio web pero ahora con las importaciones adecuadas.

Ejercicio propuesto (6)

Crear un servidor que al ser consultado devuelva un sitio web de fondo negro y letras blancas, cuyo código CSS sea obtenido de una ruta especificada en el servidor.

Resumen

A lo largo de esta lectura cubrimos:

- Creación de interfaces de línea de comandos con el Paquete Yargs.
- Trabajando imágenes con el paquete Jimp.
- Bajar una aplicación utilizando la línea de comandos.
- Devolver sitios web estáticos desde el servidor.

Solución de los ejercicios propuestos

1. Crear una interfaz de línea de comando con el paquete Yargs que defina un comando "PING" y un argumento número y devuelva el mensaje "PONG" y el número recibido concatenadamente.

```
const yargs = require('yargs')
const argv = yargs
  .command(
    'PING',
    '',
    {
      numero: {
        describe: '',
        demand: true,
        alias: 'n',
      },
    },
    (args) => {
      console.log(`PONG ${args.numero}`)
    }
  )
  .help().argv
```

2. Crear una interfaz de línea de comando con el paquete Yargs que defina un comando "adulto" y un argumento "edad" que al ser ejecutado evalúe si el valor del argumento es mayor a 18. y de ser así devolver el mensaje "Mayor de edad", de lo contrario devolver "Menor de edad"

```
const yargs = require('yargs')
const argv = yargs
  .command(
    'adulto',
    'Comprobando edades',
    {
      edad: {
        describe: 'edad',
        demand: true,
        alias: 'e',
      },
    },
    (args) => {
```

```
    args.edad > 18 ? console.log('Mayor de edad') : console.log('Menor de edad')
  }
)
.help().argv
```

3. Crear un servidor que al ser consultado devuelva una imagen procesada con los métodos `resize`, `grayscale` y `quality`.

El método `grayscale` al igual que el `sepia` no necesitan recibir un parámetro, no obstante el `quality` recibe un número del 0 al 100 indicando el porcentaje de calidad.

```
const Jimp = require('jimp')
const http = require('http')
const fs = require('fs')
http
  .createServer((req, res) => {

    Jimp.read('https://miviaje.com/wp-content/uploads/2016/05/shutterstock_337174700.jpg', (err, imagen) => {
      imagen
        .resize(550, Jimp.AUTO)
        .greyscale()
        .quality(20)
        .writeAsync('img.png')
        .then(() => {
          fs.readFile('img.png', (err, Imagen) => {
            res.writeHead(200, { 'Content-Type': 'image/jpeg' })
            res.end(Imagen)
          })
        })
    })
  })
  .listen(3000, () => console.log('Servidor corriendo bajo el PID', process.pid))
```

4. Crear un servidor que al ser consultado devuelva una etiqueta img formato String

```
const http = require('http')
http
  .createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html' })
    res.end(`
      
    `)
  })
  .listen(3000, () => console.log('Servidor encendido'))
```

5. Crear un servidor que al ser consultado devuelva el siguiente código HTML

```
<script>
  alert('Soy el archivo Alerta.html ¡Funciona!')
</script>
```

```
const http = require('http')
const fs = require('fs')
http
  .createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html' })
    res.writeHead(200, { 'Content-Type': 'text/html' })
    fs.readFile('Alerta.html', 'utf8', (err, html) => {
      res.end(html)
    })
  })
  .listen(3000, () => console.log('Servidor encendido'))
```

6. Crear un servidor que al ser consultado devuelva un sitio web de fondo negro letras blancas cuyo código CSS sea obtenido de una ruta especificada en el servidor.

```
<head>
  <link rel="stylesheet" href="http://localhost:3000/estilos" />
</head>
<p>Texto random en el archivo index.html</p>
```

```
body {
  background: black;
  color: white;
}
```

```
const http = require('http')
const fs = require('fs')
http
  .createServer((req, res) => {
    if (req.url === '/') {
      res.writeHead(200, { 'Content-Type': 'text/html' })
      fs.readFile('index.html', 'utf8', (err, html) => {
        res.end(html)
      })
    }
    if (req.url === '/estilos') {
      res.writeHead(200, { 'Content-Type': 'text/css' })
      fs.readFile('estilos.css', (err, css) => {
        res.end(css)
      })
    }
  })
  .listen(3000, () => console.log('Servidor encendido'))
```