Introducción a Node

Ciclo de vida de un proceso Node

{desafío} latam_

Describir las características fundamentales del entorno Node.js y su utilidad para el desarrollo de aplicaciones web

- Unidad 1:
  Introducción a Node
- Unidad 2:
  Node y el gestor de paquetes
- Unidad 3:
  Persistencia

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Describir las características fundamentales del entorno Node para el desarrollo de aplicaciones web.*

¿Cuáles son las diferencias principales entre una ruta y un middleware en el desarrollo de servidores con el framework Express?

{desafío} latam_

/* Child process */

Child Process

¿Qué es?

- Un "Child process" (proceso hijo) en Node.js es como un asistente o ayudante que puede ejecutar tareas por separado mientras tu programa principal sigue funcionando sin detenerse. Imagina que tienes un ayudante (proceso hijo) que te ayuda a hacer ciertas tareas mientras tú (proceso padre) te ocupas de otras cosas.

- El módulo child_process de Node.js te permite crear y comunicarte con estos "procesos hijos". Puedes decirle al proceso hijo que haga algo, y puedes recibir resultados o información de él cuando termine su tarea.

Child Process

Tarea delegadas a procesos externos

Delegar tareas a procesos externos, también conocido como "external process execution", es una estrategia en la que una aplicación o programa utiliza procesos independientes para realizar ciertas tareas en lugar de llevar a cabo esas tareas directamente dentro del proceso principal.

Esta estrategia es comúnmente utilizada en varios contextos, incluyendo:

- Tareas pesadas o intensivas en recursos
- Tareas en paralelo
- Integración con sistemas externos

Child Process

¿Cómo comunicarse?

El módulo child_process proporciona diferentes formas de crear y comunicarse con procesos hijos. Algunas de las principales funciones que se utilizan son:

1. `child_process.spawn()`
2. `child_process.exec()`
3. `child_process.execFile`
4. `child_process.fork`

A continuación, revisaremos un ejemplo de uso de child process

Demostración "Aplicando child_process"

Ejercicio guiado

Aplicando child_process

Para el primer encuentro con el módulo child_process usaremos el método “exec” para ejecutar una línea de comando indicando la siguiente sintaxis “node <nombre_del_archivo>” anteriormente expuesta con el “Hola Mundo!”.

Crea una aplicación que ejecute 2 archivos y manipule su respuesta, en este caso numéricas, para devolver la suma de ambas. Para esto sigue los siguientes pasos:

Ejercicio guiado

Aplicando child_process

- **Paso 1:** Crear un archivo con nombre “num1.js” y usa el siguiente código para mandar por consola el número 1.

```
console.log(1)
```

- **Paso 2:** Crear un archivo con nombre “num2.js” y usa el siguiente código para mandar por consola el número 2.

```
console.log(2)
```

Ejercicio guiado

Aplicando child_process

Ahora solo falta crear un tercer archivo cuyo objetivo será sumar los números devueltos por la ejecución de los otros archivos.

- **Paso 3:** Crear un archivo con nombre “suma.js”, el cual será nuestra aplicación principal. Escribe el código realizando las siguientes instrucciones:

    1. Importar el módulo “child_process” en una constante con el mismo nombre.

```
// 1
const child_process = require('child_process')
```

Ejercicio guiado

Aplicando child_process

2. Crear las variables num1 y num2 de forma global con valores null.

```
// 2
let num1 = null
let num2 = null
```

2. Crear una función “ejecutar” que reciba como parámetro el nombre de un archivo a ejecutar.

```
// 3
function ejecutar(archivo) {
```

2. La función creada retornará una promesa.

```
// 4
return new Promise((resolve) => {
```

Ejercicio guiado

Aplicando child_process

5. Usar el método “exec” pasando como primer parámetro una concatenación con el parámetro recibido en la función, y como segundo parámetro el callback “resolve” con el resultado de la ejecución en formato Number. Harás esto para poder sumar aritméticamente ambos resultados.

```
   // 5
   child_process.exec(`node ${archivo}`, function (err, result) {
     resolve(Number(result))
   })
 })
}
```

Ejercicio guiado

Aplicando child_process

6. Llamar a la función ejecutar pasando como argumento el nombre del primer archivo y en el callback de su método “then”, reasignamos el valor de la variable “num1” con el parámetro “numero1”, recibido como parámetro por el callback “resolve” de la promesa.

```
// 6
ejecutar('num1.js').then((numero1) => {
 num1 = numero1
```

Ejercicio guiado

Aplicando child_process

7. Dentro del método “then” de la primera llamada a la función, vuelve a llamar a la función ejecutar pero esta vez especificando el nombre del segundo archivo reasignando en el callback el valor de “num2” con el parámetro “numero2” e imprimiendo por consola la suma de ambas variables.

```
// 7
ejecutar('num2.js').then((numero2) => {
  num2 = numero2
  console.log(num1 + num2)
})
})
```

```
// 1
const child_process = require('child_process')

// 2
let num1 = null
let num2 = null

// 3
function ejecutar(archivo) {

 // 4
 return new Promise((resolve) => {

   // 5
   child_process.exec(`node ${archivo}`, function (err, result) {
     resolve(Number(result))
   })
 })
}
// 6
ejecutar('num1.js').then((numero1) => {
 num1 = numero1
 // 7
 ejecutar('num2.js').then((numero2) => {
   num2 = numero2
   console.log(num1 + num2)
 })
})
```

Ejercicio guiado

Aplicando child_process

- **Paso 4:** En la terminal corre el comando “node suma.js” y mira lo que sucederá en la siguiente imagen.

```
→  ejercicio lectura git:(master) ✗ node num1.js
1
→  ejercicio lectura git:(master) ✗ node num2.js
2
→  ejercicio lectura git:(master) ✗ node suma.js
3
```

Si quieres saber más del módulo child_process, su API completa la podrás conseguir de forma oficial en el siguiente link.

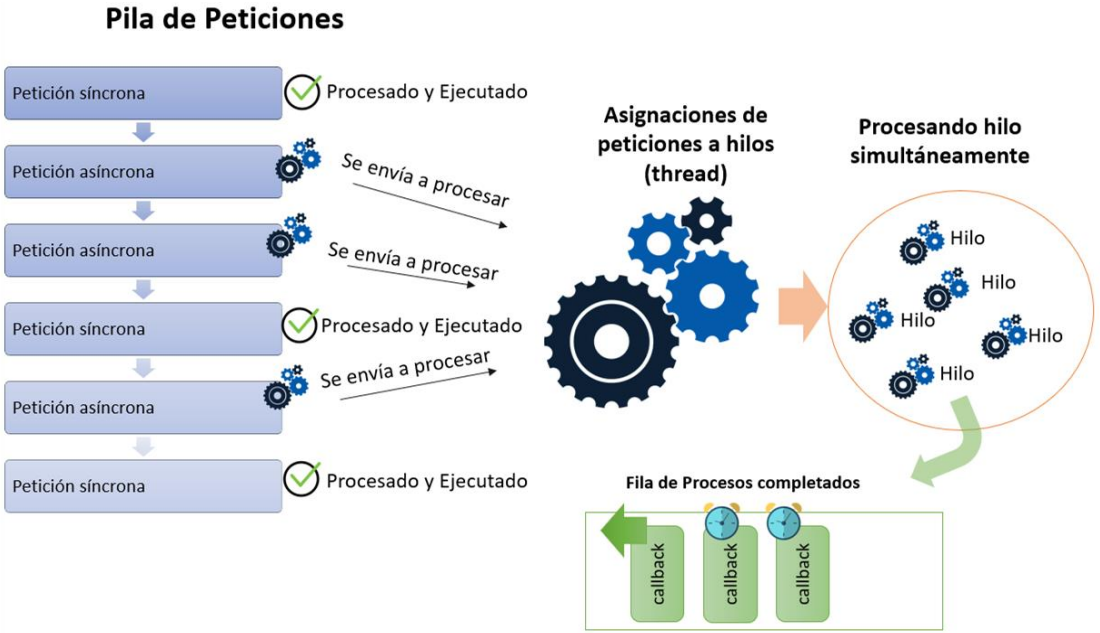
/* Ciclo de vida de un proceso Node */

Ciclo de vida de un proceso Node

¿Cómo funciona?

Una de las características que más destacan en Node es su velocidad y rendimiento en los procesos que realiza, esto se logra al manejar sus operaciones de manera asíncrona, es decir, permite responder rápidamente a una serie de peticiones o llamadas simultáneas, optimizando el rendimiento en sus procesos.

El ciclo de vida de un proceso Node por su naturaleza asíncrona, seguiría la secuencia que verás en la siguiente imagen.



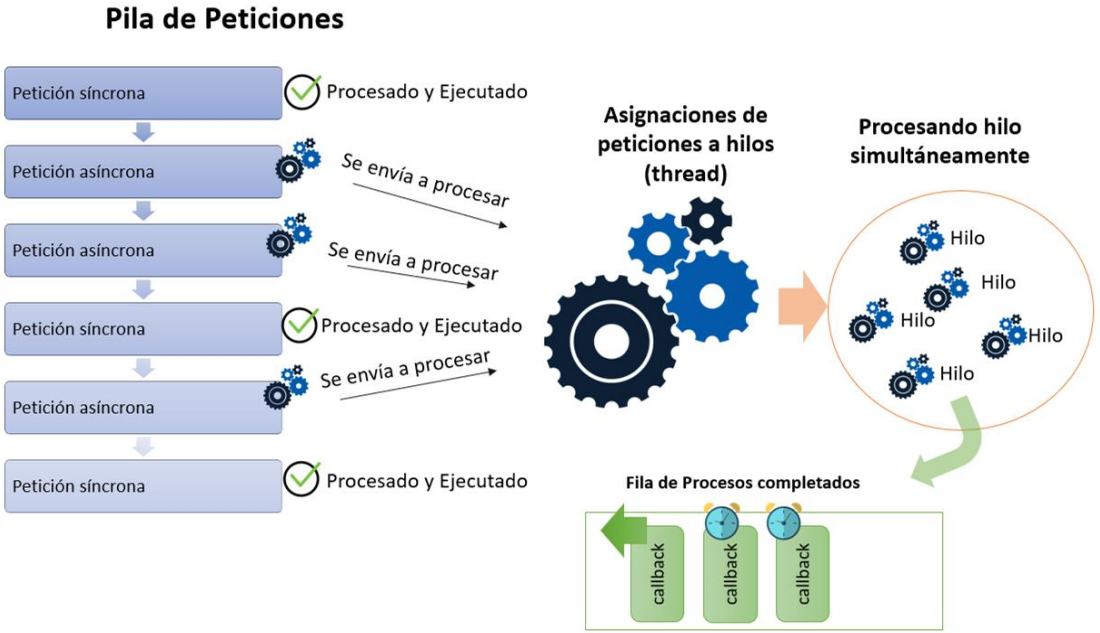
{desafío} latam_

Ciclo de vida de un proceso Node

¿Cómo funciona?

El funcionamiento detallado se describe de la siguiente manera:

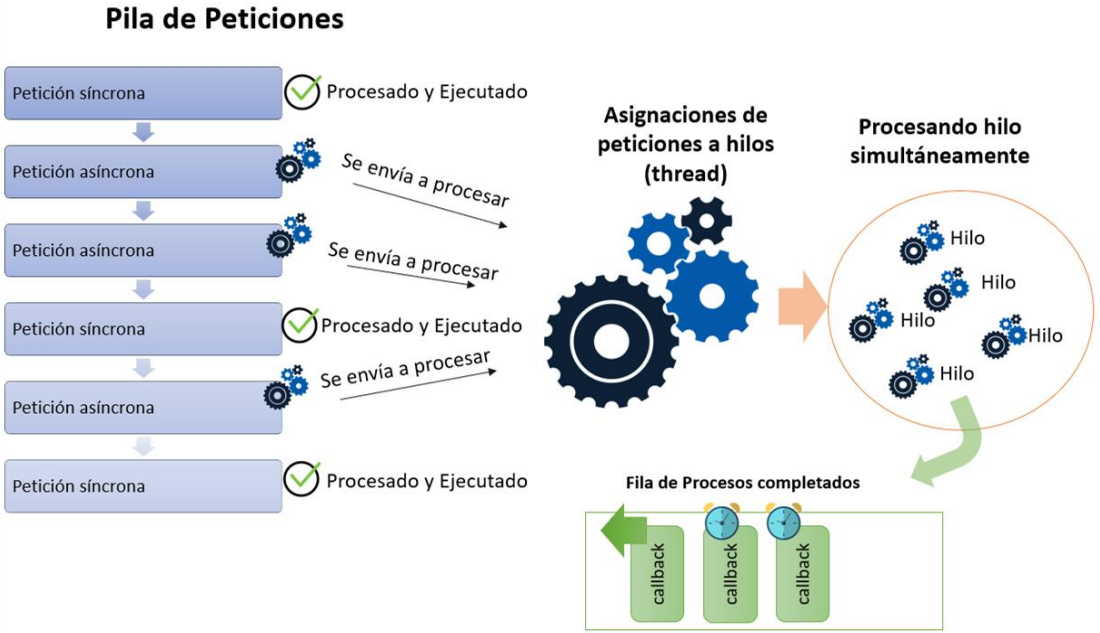
- La primera petición es síncrona, por lo tanto la ejecuta inmediatamente, deteniendo la ejecución del programa hasta finalizar y despacharla, para luego pasar a la que viene posterior, esta se demuestra con un ícono de check verde en la imagen.



Ciclo de vida de un proceso Node

¿Cómo funciona?

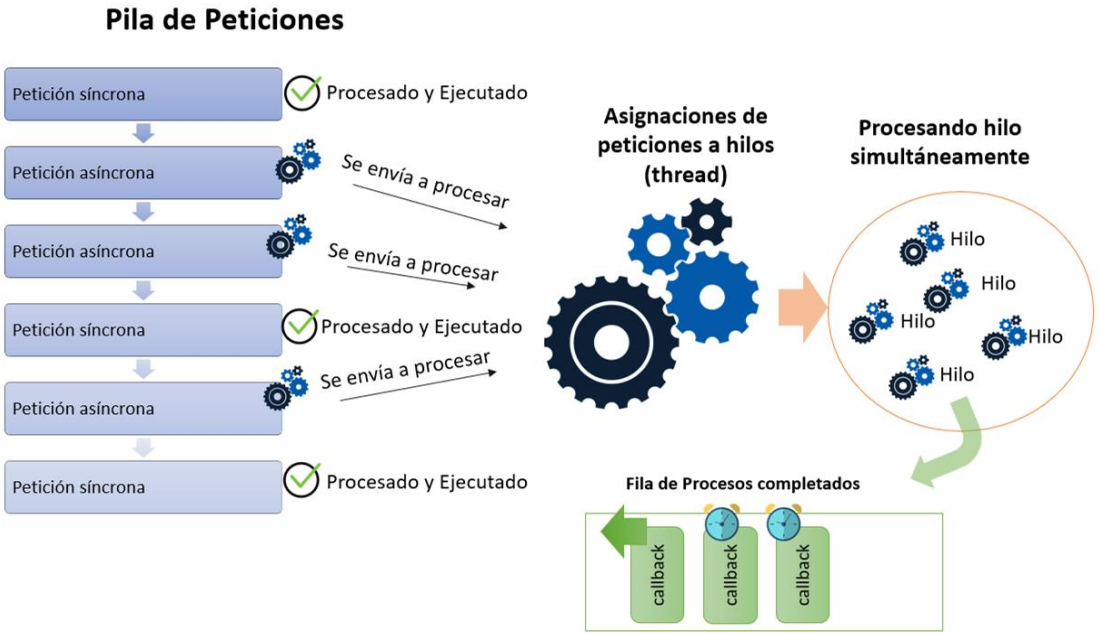
- La segunda y tercera petición son asíncronas, no las ejecuta y las envía una a una a un proceso intermediario, que la asigna a un hilo de subproceso para sus ejecuciones simultáneas, permitiendo a la pila de peticiones seguir evaluando sin detener la ejecución del programa y así continúa su evaluación hasta terminar con la pila de peticiones.



Ciclo de vida de un proceso Node

¿Cómo funciona?

- El programa termina su ejecución y queda disponible para otras tareas, sin embargo aún pueden haber procesos ejecutándose, que serían nuestros hilos de peticiones asíncronas corriendo en paralelo, las cuales una vez que completen su tarea, irán acumulándose como funciones callback en una fila, despachándose en orden de llegada.



Ciclo de vida de un proceso Node

Peticiones asíncronas y threads

Comencemos por conocer las definiciones de estos conceptos:

- **Petición Asíncrona:** Es una operación que mientras se encuentra en ejecución, no bloquea las operaciones que puedan venir o las que ya están en ejecución, permitiendo que se sigan procesando de manera simultánea todas las operaciones.
- **Petición Síncrona:** Es una operación que mientras se encuentra en ejecución, no permite que las operaciones posteriores se ejecuten hasta finalizar su proceso.
- **Thread:** Su traducción al español es “hilo”. Se define como un conjunto de pequeñas tareas encadenadas para ser ejecutadas por el sistema operativo, además los hilos o conjuntos de tareas utilizan recursos del sistema operativo, que pueden estar siendo utilizados por otros hilos, y a su vez se conocen como procesos. Un proceso permanece en ejecución, hasta que cada uno de sus hilos o tareas haya finalizado.

Ciclo de vida de un proceso Node

Peticiones asíncronas y threads

Node se caracteriza por su **velocidad** y **rendimiento**, sabemos que para esto debe manejar sus operaciones de manera especial.

- En un programa, las instrucciones o peticiones que vamos indicando se irán evaluando de forma secuencial, de manera que si es una petición síncrona el programa esperará hasta que el proceso culmine para continuar con la lectura del siguiente proceso

- Luego continúa con la evaluación de la petición posterior. Si la petición es asíncrona en este caso, no la ejecuta y la envía a un proceso que la asigna a un subproceso o hilo, y continúa con la evaluación de las siguientes peticiones.

- Es así como Node optimiza su rendimiento y velocidad de respuesta, segmentando por tipo de peticiones y realizando ejecución de tareas simultáneas con las peticiones asíncronas.

Ciclo de vida de un proceso Node

Callbacks - Instrucciones blocking v/s non-blocking

- En JavaScript, los callbacks son funciones que se utilizan para manejar tareas asincrónicas. Son un concepto fundamental en programación asíncrona y permiten que ciertas operaciones se realicen después de que otras hayan finalizado, evitando bloquear la ejecución del programa mientras esperan una respuesta.

- En términos sencillos, un callback es una función que se pasa como argumento a otra función y se ejecuta después de que la función original haya terminado su trabajo. Esto es especialmente útil cuando se trabaja con operaciones que llevan tiempo, como leer archivos, hacer solicitudes a servidores o realizar animaciones.

Ciclo de vida de un proceso Node

Callbacks - Instrucciones blocking v/s non-blocking

Las instrucciones blocking y non-blocking están altamente enlazadas a los callbacks y a la asincronía.

- Una instrucción **blocking** se considera aquella que detiene la lectura de la siguiente instrucción, puesto que puede resolverse casi inmediatamente sin dependencias externas a ella misma.

- Las instrucciones **non-blocking** son ejecutadas en paralelo al hilo de instrucciones y en los casos donde la asincronía existe por consultar recursos externos, se sufre de la incertidumbre de no saber cuánto tiempo necesitarán para resolverse.

Demostración "Callback"

Ejemplo
Callback

Ejemplo sencillo generando asincronía con el método setTimeout:

- **Paso 1:** Crear una función llamada “holaMundo”.
- **Paso 2:** Dentro de la función mandamos por consola el mensaje “Hola Mundo!”.
- **Paso 3:** Crear un timer con el método setTimeout que ejecute la función creada luego de 3 segundos.

```
// 1
function holaMundo()
{
// 2
 console.log("Hola mundo!");
}
// 3
setTimeout(holaMundo, 3000);
console.log("Nos vemos
mundo!");
```

¿Cuál es el propósito del módulo "child_process" en Node.js?

¿Cuál es el papel de los callbacks en el funcionamiento asíncrono de Node.js y cómo contribuyen a su rendimiento y velocidad en la ejecución de peticiones?

{desafío} latam_

Próxima sesión...

- *Desafío - Abracadabra*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales