Sesión # 11 Componente Práctico

Desarrollo de Back-end web con Node.js

Vamos a repasar un poco los conceptos aprendidos en clase. Realiza los siguientes ejercicios prácticos:

EventEmitter

Como ya conoces, parte de la interacción del usuario se maneja a través de eventos: clics del mouse, pulsaciones de botones del teclado, reacción a los movimientos del mouse, etc. En el lado del backend, Node.js nos ofrece la opción de construir un sistema similar usando el events module.

Este módulo, en particular, ofrece la clase EventEmitter, que usaremos para manejar nuestros eventos.

Ejercicio 1

1. Crea un nuevo archivo app.js para tu solución
2. Inicializa los módulos y la clase EventEmitter

   ```
   const EventEmitter = require("events");
   const eventEmitter = new EventEmitter();
   ```

   Esta clase expone, entre muchos otros, los métodos on y emit.

   `emit` se utiliza para desencadenar un evento
   `on` se usa para agregar una función de devolución de llamada que se ejecutará cuando se active el evento

3. Ahora, crea un evento ‘start’, y como respuesta, reacciona a él simplemente registrando en la consola la palabra started, sigue el siguiente código:

   ```
   eventEmitter.on('start', () => {
     console.log('started')
   })
   ```

4. Por último, añade la ejecución del evento
   `eventEmitter.emit('start')`

   Esto activará la función del controlador de eventos y obtendremos el registro de la consola.

5. Ahora corre tu ejemplo, ejecuta:
   `npm install` y luego, `node app.js`

Información de interés - Puede pasar argumentos al controlador de eventos enviandolos como argumentos adicionales a emit(), ejemplo:

```
eventEmitter.on('start', (start, end) => {
 console.log(`started from ${start} to ${end}`)
})
eventEmitter.emit('start', 1, 100)
```

Callbacks

Un callback (llamada de vuelta) es el equivalente asíncrono a una función. Se llama a una función de devolución de llamada al completar una tarea determinada.

Por ejemplo, una función para leer un archivo puede comenzar a leer el archivo y devolver el control al entorno de ejecución inmediatamente para que se pueda ejecutar la siguiente instrucción. Una vez que se completa la Escritura (E) / Subida (S) del archivo, se llamará a la función de callback y mientras esto ocurre se envía a la misma función el contenido del archivo como parámetro. Por lo tanto, no hay bloqueo ni espera a la E / S de archivos. Esto hace que Node.js sea altamente escalable, ya que puede procesar una gran cantidad de solicitudes sin esperar a que ninguna función devuelva resultados.

Ejercicio 2 (con bloqueo):

1. Cree una nueva carpeta para la solución de este ejercicio
2. Dentro de la carpeta, cree un archivo de texto llamado input.txt con el siguiente contenido:
   `Esto es un ejemplo para MisionTIC 2022!!!!!`

3. Ahora, cree un archivo js llamado blocked.js con el siguiente código:

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("Program Ended");
```

4. Ejecute blocked.js para ver el resultado:
   `$ node blocked.js`

5. Verifique la salida.

   ```
   Esto es un ejemplo para MisionTIC 2022!!!!!

   Program Ended
   ```

Ejercicio 3 (sin bloqueo):

1. Dentro de la misma carpeta, cree un archivo js llamado no-blocked.js con el siguiente código:

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

fs.readFile("input.txt", function (err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});

console.log("Program Ended");
```

2. Ahora, ejecute el archivo no-blocked.js para ver el resultado:
   `$ node no-blocked.js`

3. Verifique la salida

   ```
   Program Ended

   Esto es un ejemplo para MisionTIC 2022!!!!!
   ```

Como puede observar, el primer ejemplo muestra que el programa se bloquea hasta que lee el archivo y luego solo procede a finalizar el programa, mientras que el segundo ejemplo muestra que el programa no espera la lectura del archivo y procede a imprimir "Programa finalizado" y al mismo tiempo, el programa sin bloqueo continúa leyendo el archivo.

Promises

Manejar tareas asíncronas en Javascript puede volverse muy complicado, sin embargo con la llegada de las promesas (promises) es posible simplificar mucho el trabajo.

Antes de la llegada de las promises a Javascript, el tratamiento de procesos asíncronos se hacía exclusivamente por medio de callbacks, que al tener que encadenar uno tras otro, el código se vuelve difícil de entender y mantener además se crea el llamado “Callback Hell”.

Como sugiere su nombre, es una promesa de que un objeto JavaScript eventualmente devolverá un valor o un error.

Una promesa tiene 3 estados:

- Pendiente: el estado inicial que indica que la operación asincrónica no está completa.
- Cumplido: significa que la operación asincrónica se completó correctamente.
- Rechazado: significa que la operación asincrónica falló.

Sintaxis básica de una promesa:

```
someAsynchronousFunction()
    .then(data => {
        // After promise is fulfilled
        console.log(data);
    })
    .catch(err => {
        // If promise is rejected
        console.error(err);
    });
```

Las promesas son importantes en JavaScript moderno, ya que se usan con async/await lo que hace que ya no sea necesario utilizar devoluciones de llamada o then() y catch() para escribir código asíncrono.

```
try {
    const data = await someAsynchronousFunction();
} catch(err) {
```

```
    // If promise is rejected
    console.error(err);
}
```

Ejercicio 4

1. Para practicar, convierte el modelo de callback utilizado en el ejercicio 3 (sin bloqueo) en uno basado en promesas.

   Ten en cuenta que la mayoría de las funciones asincrónicas que aceptan una devolución de llamada en Node.js, como el **fs (file system) module** (sistema de archivos), tienen un estilo de implementación estándar: la devolución de llamada se pasa como último parámetro.

   Por ejemplo, así es como puede leer un archivo usando fs.readFile():

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

       // Data is a buffer
       console.log(data);
   });
   ```

   El callback, que se pasa a la función, debe aceptar un Error como primer parámetro. Después de eso, puede haber cualquier número de salidas.

   Si la función que necesitas convertir en una Promesa sigue estas reglas, puedes utilizar **util.promisify**, un módulo nativo de Node.js que convierte las llamadas de retorno en Promesas.

   Para ello, primero importa el módulo util:

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

   Luego se utiliza el método promisify para convertirlo en una promesa:

   ```
   const fs = require('fs');
   const readFile = util.promisify(fs.readFile);
   ```

Ahora utiliza la función recién creada como una promesa regular:

```
readFile('./sample.txt', 'utf-8')
    .then(data => {
        console.log(data);
    })
    .catch(err => {
        console.log(err);
    });
```

Alternativamente, puede utilizar async/await como se indica en el siguiente ejemplo:

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

const readFile = util.promisify(fs.readFile);

(async () => {
  try {
    const content = await readFile("../input.txt", "utf-8");
    console.log(content);
  } catch (err) {
    console.error(err);
  }
})();

console.log("Program Ended");
```

Por otro lado, es posible crear la promesa sin utilizar el metodo util.promisify. Para ello, envuelve el servicio que usa callbacks en otra función que retorne una promesa. Y haz el llamado de resolve con los datos que retorna el callback del servicio.

Sintaxis básica de una promesa:

```
new Promise(function (resolve, rejection) {
})
```

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

const readFile = (fileName) => {
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, (err, data) => {
      if (err) {
        return reject(err);
      }

      resolve(data.toString());
    });
  });
};

readFile("../input.txt")
  .then((data) => {
    console.log(data.toString());
  })
  .catch((err) => {
    console.log(err);
  });
console.log("Program Ended");
```