



El futuro digital  
es de todos

MinTIC



# CICLO IV:

## Desarrollo de Aplicaciones Web

Mision  
TIC2022



El futuro digital  
es de todos

MinTIC

**UN** UNIVERSIDAD  
DEL NORTE

Vigilada Mineducación

# Sesión 11: Desarrollo de Aplicaciones Web

Desarrollo de Back-End web con Node.js

Mision  
TIC2022



# Objetivos de la sesión

Al finalizar esta sesión estarás en capacidad de:

1. Implementar funcionalidades base de Node.js en una aplicación web.
2. Gestionar paquetes de Node.js.



# Node.js - Event Emitter

- Así como los navegadores web cuenta con un control de eventos como **onclick**, **onmouseenter**, **onmouseleave**, entre otros. Node.js cuenta su propio sistema de eventos.
- Este sistema de eventos se conoce como el Event Emitter.
- Para escuchar a un evento solo debemos de agregar el siguiente código:

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

- Donde definimos un evento start, el cual recibe como parámetro una variable number.
- Luego para disparar o ejecutar el evento lo hacemos de la siguiente forma:

```
eventEmitter.emit('start', 23)
```



# Node.js - Event Emitter

- Cabe resaltar que se puede definir un evento con varios parámetros de la siguiente forma:

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

- Luego para disparar o ejecutar el evento lo hacemos de la siguiente forma:

```
eventEmitter.emit('start', 1, 100)
```

- Ejemplo adaptado de [Node.js](#).



# Node.js - Callback Hell

- Inicialmente en JS no existían las **Promises** ni, las arrow functions.
- Por lo que para manejar código que se ejecutara de forma asíncrona se utilizaban callbacks.
- Estos callbacks constan básicamente de un método que es pasado como argumento a otro método que realiza una tarea asíncrona.
- Esto con el fin de ejecutar el callback justo después de que el método asíncrono termine su ejecución.



# Node.js - Callback Hell

- A modo de demostración consideremos el siguiente escenario:
  - Se dispara un evento de Node.js
  - Requerimos leer un archivo.
  - Requerimos enviar una solicitud HTTPs.
  - Requerimos modificar el archivo previamente existente.



# Node.js - Callback Hell

- El escenario previamente descrito se puede implementar de la siguiente forma, primero hablemos de nuestros métodos “asíncronos”:

```
function readFile(success, error) {  
  const hasError = Math.random() > 0.9;  
  const errorData = {};  
  if (hasError) error(errorData);  
  
  console.log('Leyendo archivo');  
  const data = {};  
  setTimeout(function(){  
    success(data);  
  }, 1000);  
}
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.





# Node.js - Callback Hell

- Así mismo para el envío de nuestra “solicitud HTTP”:

```
function sendHTTP(data, success, error) {  
  const hasError = Math.random() > 0.95;  
  const errorData = {};  
  if (hasError) error(errorData);  
  
  console.log('Enviado solicitud HTTP');  
  const response = {};  
  setTimeout(function(){  
    success(response);  
  }, 2000);  
}
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.



# Node.js - Callback Hell

- Así mismo para “la escritura de nuestro archivo”:

```
function writeFile(reponse, success, error) {  
  const hasError = Math.random() > 0.95;  
  const errorData = {};  
  if (hasError) error(errorData);  
  
  console.log('Escribiendo Archivo');  
  const data = {};  
  setTimeout(function(){  
    success(data);  
  }, 2000);  
}
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.



# Node.js - Callback Hell

- Y luego nuestro controlador de error sería el siguiente:

```
function errorHandler(error) {  
  console.error(error);  
}
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.



# Node.js - Callback Hell

- Finalmente este código será ejecutado de la siguiente forma:

```
readFile(  
  function (data) {  
    sendHTTP(  
      data,  
      function (response) {  
        writeFile(  
          response,  
          function (message) {  
            console.log('Archivo escrito');  
          },  
          function (error) { errorHandler(error); }  
        );  
      },  
      function (error) { errorHandler(error); }  
    );  
  },  
  function (error) { errorHandler(error); }  
);
```

- Esto es lo que se conoce como callback hell dado que es muy difícil de leer.



# Node.js - Callback Hell

- Esto ocurre porque nuestros eventos al ser asíncronos no hay forma secuencial de escribir el código línea a línea manteniendo su contexto.
- Por lo que como solución se ideó que los métodos asíncronos recibieran dos callbacks:
  - **success**: En caso de todo ser ejecutado como es esperado se continúa la ejecución de nuestro programa con este callback
  - **error**: Cuando ocurra alguna excepción o error en nuestro programa, se llama este callback para controlarla y se termina su ejecución.
- Esto permite que nuestros callbacks puedan recibir información de nuestros métodos asíncronos.
- Para ver el ejemplo tenemos el archivo de [Stackblitz](#).



# Node.js - Promises

- Como medida en contra el callback hell se idearon las Promises y las arrow function.
- Por lo que podríamos reescribir nuestro escenario de la siguiente forma:

```
const readFile = () =>
  new Promise((resolve, reject) => {
    const hasError = Math.random() > 0.9;
    const errorData = {};
    if (hasError) reject(errorData);

    console.log('Leyendo archivo');

    const data = {};

    setTimeout(() => resolve(data), 1000);
  });
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.



# Node.js - Promises

- Así mismo para el envío de nuestra “solicitud HTTP”:

```
const sendHTTP = (data) =>
  new Promise((resolve, reject) => {
    const hasError = Math.random() > 0.95;
    const errorData = {};
    if (hasError) reject(errorData);

    console.log('Enviado solicitud HTTP');

    const response = {};

    setTimeout(() => resolve(response), 2000);
  });
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.



# Node.js - Promises

- Así mismo para “la escritura de nuestro archivo”:

```
const writeFile = (reponse) =>
  new Promise((resolve, reject) => {
    const hasError = Math.random() > 0.95;
    const errorData = {};
    if (hasError) reject(errorData);

    console.log('Escribiendo Archivo');

    const data = {};

    setTimeout(() => resolve(data), 2000);
  });
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.





# Node.js - Promises

- Y luego nuestro controlador de error sería el siguiente:

```
const errorHandler(error) => console.error(error);
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.



# Node.js - Promises

- Finalmente este código será ejecutado de la siguiente forma:

```
readFile()  
  .then((data) => sendHTTP(data))  
  .then((response) => writeFile(response))  
  .then(() => console.log('Archivo escrito'))  
  .catch((error) => errorHandler(error));
```

- Si comparamos el uso de Promises con el de callback Hell notamos de forma inmediata que este es más sencillo de leer y escalar.
- Pero de igual forma si necesitáramos hacer algo con los datos de lectura al final de nuestro proceso tendríamos problemas puesto que no se mantiene el contexto de estos datos.
- Razón por la cual no tenemos acceso a ellos.



# Node.js - Async / Await

- Para ello se introducen los keywords `async` / `await`.
- Tomando esto en cuenta podemos reescribir nuestro escenario de la siguiente forma:

```
const readFile = async () => {  
  const hasError = Math.random() > 0.9;  
  const errorData = {};  
  if (hasError) throw new Error(errorData);  
  
  console.log('Leyendo archivo');  
  const data = {};  
  
  await delay(1000); // simulamos una actividad asincrona  
  return data;  
};
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.



# Node.js - Async / Await

- Así mismo para el envío de nuestra “solicitud HTTP”:

```
const sendHTTP = async (data) => {  
  const hasError = Math.random() > 0.95;  
  const errorData = {};  
  if (hasError) throw new Error(errorData);  
  
  console.log('Enviado solicitud HTTP');  
  const response = {};  
  
  await delay(2000); // simulamos una actividad asincrona  
  return response;  
};
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.



# Node.js - Async / Await

- Así mismo para “la escritura de nuestro archivo”:

```
const writeFile = async (reponse) => {  
  const hasError = Math.random() > 0.95;  
  const errorData = {};  
  if (hasError) throw new Error(errorData);  
  
  console.log('Escribiendo Archivo');  
  const data = {};  
  
  await delay(2000); // simulamos una actividad asíncrona  
  return data;  
};
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.



# Node.js - Async / Await

- Y luego nuestro controlador de error sería el siguiente:

```
const errorHandler(error) => console.error(error);
```

- Adicionalmente tenemos un método para simular una espera asíncrona:

```
const delay = (ms) => new Promise((resolve) => setTimeout(resolve, ms));
```

- Cabe resaltar que estos métodos son a modo de demostración y no cumplen una funcionalidad real.



# Node.js - Async / Await

- Finalmente este código será ejecutado de la siguiente forma:

```
const proccess = async () => {  
  try {  
    const data = await readFile();  
    const response = await sendHTTP(data);  
    await writeFile(response);  
  
    console.log('Archivo escrito');  
  } catch (error) {  
    errorHandler(error);  
  }  
};  
  
proccess();
```

- De esta forma tenemos un archivo mucho más legible y escalable.
- Así mismo, ya tenemos acceso a toda la información obtenida de todos nuestros procesos asíncronos en caso de ser requerida.

- Ver [Stackblitz](#).



# Node.js - Ecosistema

- Sabemos que Node.js cuenta con un repositorio de paquetes open-source llamado npm.
- También contamos con un CLI llamado npm para la instalación de estos paquetes.
- Para instalar un paquete solo debemos de ejecutar el comando **npm install <nombre de paquete>**.
- Esto nos creará el folder `node_modules/` donde encontraremos todos los paquetes instalados.
- Si observamos la configuración de “dependencies” en nuestro `package.json`, observaremos que se almacena un listado de los paquetes instalados con su respectiva versión.
- En caso que clonemos un proyecto de node, necesitaremos instalar los paquetes con el comando **npm install**.





# Node.js - Ecosistema

- Para utilizar paquetes, instalaremos dos paquetes a modo de demostración:
  - `lodash`.
  - `esm`.
- Importamos nuestros paquetes de la forma tradicional (CommonJS) y de la forma moderna (ModuleJS):
  - Tradicional: `const _ = require('lodash');`
  - Moderna: `import * as _ from 'lodash';`
- Cabe resaltar que para usar la forma moderna hay que instalar el paquete `esm` o habilitar flags experimentales de Node.js o utilizar un **transpilador** como Babel.
- Ver [require.js](#) e [import.js](#) en [Stackblitz](#).



El futuro digital  
es de todos

MinTIC

**UN** UNIVERSIDAD  
**DEL NORTE**

Vigilada Mineducación

# Ejercicios de práctica

Mision  
TIC2022



# Referencias

- <https://nodejs.dev/learn/the-nodejs-event-emitter>
- <https://www.npmjs.com/>



El futuro digital  
es de todos

MinTIC

**UN** UNIVERSIDAD  
**DEL NORTE**

Vigilada Mineducación

**¡GRACIAS**  
**POR SER PARTE DE**  
**ESTA EXPERIENCIA**  
**DE APRENDIZAJE!**



**Misión**  
**TIC 2022**