Getting started with Node

We're using node 18.x LTS

Hola!

As tradition dictates, we absolutely must start with a hello world program. Create a simple hello world in node:

1. Create a folder
2. Create a file named index.js in that folder
3. Code as follows:

index.js

```
// run from terminal
console.log('Hello node world!');
```

From the terminal:

```
$ node index.js
```

…watch the magic happen!

Simple file read / write

Create a folder named files and a file named write .js with the following contents:

write.js

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

// Synchronously write to a file
// This will block the event loop until the file is written
try {
    fs.writeFileSync('example.txt', 'Hello, world!');
```

write.js

```
    console.log('File written successfully');
} catch (err) {
    console.error('Error writing file:', err);
}
```

Run the file and notice a file named `example.txt` gets created.

Now let's try to read it. Create a file named read.js as follows:

read.js

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

// Synchronously write to a file
// This will block the event loop until the file is written
try {
    fs.writeFileSync('example.txt', 'Hello, world!');
    console.log('File written successfully');
} catch (err) {
    console.error('Error writing file:', err);
}
```

In this example, `fs.writeFileSync` and `fs.readFileSync` are used for writing to and reading from a file, respectively. These methods block the event loop until they complete their operations, which is not ideal for I/O-heavy applications but can be simpler for understanding basic file operations.

Asynchronous file read / write

Create a folder named `node_files` and a file named `write.js` as follows:

write.js

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

write.js

```
fs.writeFile("test01.txt", "This is line 1,\nThis is line 2", (err)=>{
    if (err) throw err;
    console.log('File written successfully!');
});
```

Run the file and you'll notice a file named `test01.txt` has been created. Now to read the contents of the file, create read.js file and run it:

index.js

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

fs.readFile('test01.txt', 'utf-8', (err, data) =>{
    if (err) throw err;
    console.log(data);
});
```

When you run the script, you'll notice the contents of the file outputted to the screen.

In the asynchronous example, fs.writeFile and fs.readFile are used. These functions take a callback that is called when the operation completes. This pattern doesn't block the event loop, allowing Node.js to handle other tasks in the meantime.

Explanation

Synchronous Operations: Simple but can block your application, making it unresponsive until the file operation is complete.

Asynchronous Operations: More complex due to callbacks, but they allow your application to remain responsive. They are the preferred way of handling I/O in Node.js.

Enhancing the “reader”

What I would like to do is create a slightly more advanced reader in my node JS. I would like to incorporate the following features:

1. Incorporate ES standard
2. Get the filename from the CLI argument
3. Put in some error checks for:
   3.1. Whether the CLI argument has been passed. If not, make a colorful suggestion.
   3.2. See whether I come across any hurdles in trying to read the file and handle the error in a colorful and user friendly manner.
4. Exit gracefully!

Now the thing about ES standard is that it uses import instead of require for the dependencies. Also, you need an initiation of the node project. So here is what I’s do:

1. Create a folder and navigate to it:
   `$ mkdir reader`
   `$ cd reader`
2. Run the node initiator using the Node Package Manager (npm):
   `$ npm init -y`
   I went along with the defaults
3. In the package .json file, add the lines:
   `"type" : "module"` right after the `"main" :....` part
4. Create a file named reader and run it:

reader.js

```
// const fs = require("fs");
// const path = require("path");
import fs from "fs"; // ES compliance
import path from "path"; // ES compliance

// Get the filename from the command line arguments
const fileName = process.argv[2];

//colors
const red = '\x1b[31m'; // ANSI code for red
const green = '\x1b[32m'; // ANSI code for green
const reset = '\x1b[0m'; // ANSI code to reset color

// making sure that a filename has been enterred
```

reader.js

```
if (!fileName) {
    console.error("\nPlease provide a file name as an argument.");
    console.error(`\nUse the format: \n   ${green}${path.basename(process.argv[0])} ` +
            `${path.basename(process.argv[1])} [your_filename.extension]${reset}\n`);
    process.exit(1);
}

fs.readFile(fileName, 'utf-8', (err, data) => {
    if (err) {
        //console.error("Error reading file:", err);

        // Check for common error types
        if (err.code === 'ENOENT') {
            console.error(`\nFile ${red}${err.path}${reset} not found. Please check the filename.\n`);
        } else if (err.code === 'EACCES') {
            console.error("Permission denied. Please check your file permissions.");
        } else {
            console.error("An unknown error occurred.");
        }
        return; // Exit the callback
    }
    console.log(data);
});
```

Unless you pass a filename, it won't run. You'll also notice that it makes colorful console outputs.

More on Asynchronous operations

In Node.js, `async/await` and Promises are fundamental concepts for handling asynchronous operations. They allow you to write asynchronous code in a more synchronous-looking style, which can make it easier to read and maintain. Let's explore these concepts with an example.

Promises

A Promise in JavaScript is an object representing the eventual completion or failure of an asynchronous operation. It can be in one of three states:

- Pending: The initial state; neither fulfilled nor rejected.
- Fulfilled: The operation completed successfully.
- Rejected: The operation failed.

Here's a simple example of using a Promise:

index.js

```
function asyncOperation() {
    return new Promise((resolve, reject) => {
        // Simulate an asynchronous operation using setTimeout
        setTimeout(() => {
            const result = 'Operation completed';
            resolve(result); // Resolve the Promise with the result
        }, 1000);
    });
}

asyncOperation().then(result => {
    console.log(result); // This will be executed after the Promise is resolved
}).catch(error => {
    console.error(error); // This will be executed if the Promise is rejected
});

console.log('while waiting...')
```

In this example, `asyncOperation` returns a Promise that resolves after 1 second. The `.then()` method is used to specify what should happen once the Promise is resolved, and `.catch()` is used for error handling.

`async`/`await`

`async` and `await` are extensions of Promises and make asynchronous code even more readable by allowing you to write it in a synchronous style. Here's how you can use `async`/`await` with the previous example:

index.js

```
// Add this new code to demonstrate async / await

async function performAsyncOperation() {
    try {
        const result = await asyncOperation(); // Wait for the Promise to resolve
        console.log(result);
    } catch (error) {
        console.error(error); // Error handling if the Promise is rejected
    }
}

performAsyncOperation();

console.log('no need to wait for this either......')
```

In this example, `performAsyncOperation` is an `async` function, which means it can contain `await` expressions. The `await` keyword is used to pause the execution of the function until the Promise returned by `asyncOperation` is resolved. If the Promise is rejected, the error will be caught by the `try...catch` block.

Explanations

Promises: Provide a way to handle the eventual completion or failure of asynchronous operations. They simplify asynchronous code by avoiding deeply nested callbacks (callback hell).

async/await: Syntactic sugar on top of Promises that allows writing asynchronous code in a more synchronous-looking manner, making it cleaner and easier to understand.

These concepts are crucial for Node.js development, especially when dealing with I/O operations, network requests, or any task that takes time to complete and should not block the main thread.

index.js

```
// run from terminal
```