Node.js
Blocking & NonBlocking
Introduction

Synchronous code

```
1  const filesystem=require('fs');
2  // readFileSync takes as arguments the filepath and the character encoding
3  var textread= filesystem.readFileSync('./txt/filetoread.txt','utf8');
   console.log(textread);
```

→ Synchronous code → blocking code

- This means that each statement is processed one after the other
- So each line waits for the result of the previous one
- Thus each line blocks the execution of the rest of the code

Asynchronous code

- Asynchronous code allow us to transfer heavy work in the background, in order for the rest of the code to continue being executed
- Asynchronous code - > non-blocking code
- Lets change file reading code to asynchronous code…

from Synchronous to Asynchronous

- **fs.readFile() method syntax**:

- fs.readFile(filename, encoding, callback_function)

- **filename**: holds the name/path of the file to read
- **encoding**:holds the encoding of file (default :'utf8')
- **callback_function**: a function that is called **after** reading of file. It takes two parameters:
  - err: If any error occurs
  - data: Contents of the file.

callback_function

- callbacks for some are considered the foundation of Node.js
- In Node.js we are able to use callback functions in order to implement **asynchronous** behavior
- A callback -> is a function **called at the completion of a given task**
  - any blocking is prevented that way
  - it allows other code to run in the meantime

callback_function

- The general idea is that the callback is the last parameter (in a method or function)
- it gets called after the function is done with all operations.
- **Usually** the **first** parameter of the callback is ->error value.
- If a callback has no error then **error param is null** and the rest is/are the return value(s).

Asynchronous code

read_asynch.js > ...

```
1   const filesystem=require('fs');
2
3   // simplest way to read a file in Node.js is to use the fs.readFile() method,
4   // passing it the file path, encoding and a callback
5   //function that will be called with the file data (and the error):
6   filesystem.readFile('./txt/filetoread.txt', 'utf8', (err, data)=>{
7
8       // Display the file content
9       console.log(data);
0       console.log(err);
1   });
2
3   console.log('readFile called');
```

Asynchronous code

- Question: When we execute the code above, which log do we expect to see first?

Asynchronous code

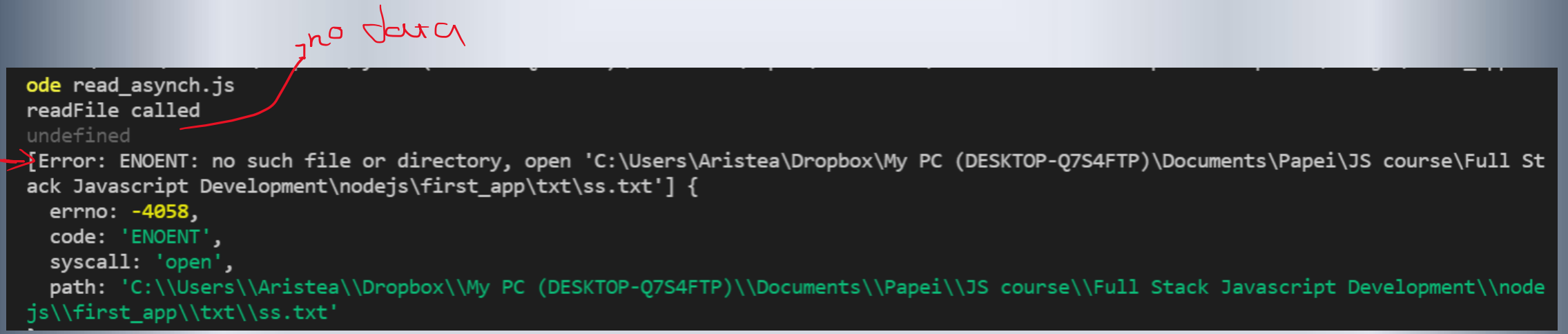
```
ull Stack Javascript Development\nodejs\first_app> node read_asynch.js
readFile called

 I will be your teacher for this course! Yeah!
null -> no error
```

So the file is being read in the background and then, immediately execution is moved on to the next statement, printing to the console

Once the data is read-> **callback** function will get called to be executed in the main single thread

Asynchronous code error example



Why is this important…

- Node.js is single threaded ->each application runs in a single thread
- All user accessing the application they access the same thread
  //not like php for example that each user has a different thread
- When a users blocks a thread with synchronous code all users need to wait for the code to be executed
- Imagine thousands of users….

Why is this important...

- So Node.js to avoid that, we use asynchronous, non-blocking code.
- In asynchronous code -> we **upload heavy work** to be worked on in the **background**
  - once that work is done -> a **callback function** is called to handle the result.
- During all that time…. rest of the code can still be executed (thus no blocking by a heavy task)

Why is this important...

- Let's consider a case where each request to a web server takes 50ms to complete and **45ms** of that 50ms is database I/O that can be done asynchronously.

- Choosing non-blocking asynchronous operations **frees up that 45ms per request** to handle other requests. This is a significant difference in capacity just by choosing to use non-blocking methods instead of blocking methods.

- SOURCE: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

Why is this important...

- The NodeJs event loop is a single thread

- **But** when this single thread encounters blocking i/o => it will delegate the task to a separate pool of worker threads.

Embedded async code

return stops execution

first_app > JS read_asynch_embedded.js > ...

```
1   const filesystem=require('fs');
2   // readFileSync takes as arguments the filepath and the character encoding
3
4   filesystem.readFile('./txt/not_embeddd.txt', 'utf8', (err, data)=>{
5       if(err) return console.log("oups!!!")
6       //not_embedded.txt contains the word embedded, that is the name of the second file
7       filesystem.readFile(`./txt/${data}.txt`, 'utf8', (err, data1)=>{
8
9           // Display the file content
10          console.log(data1);
11          console.log(err);
12      });
13
14  });
15
16  console.log('Reading file....');
```

Embedded async code

```
Reading file....
Hello there!
I am the embedded!
I hope everything is understandable, have a nice day!
null
```

Lets see how we can perform the following:

- Create directory with asyncronous function :

```
const fs = require('fs');
fs.mkdir('newdir', (err)=>{
    if(err){
        console.log('failed to create directory');
        return console.error(err);
    }else{
        console.log('Directory created successfully');
    }
});
```

To be continued…

https://nodejs.org/api/fs.html