codecademy

Node.js Essentials

The events Module

Node.js has an `EventEmitter` class which can be accessed by importing the `events` core module by using the `require()` statement. Each event emitter instance has an `.on()` method which assigns a listener callback function to a named event. `EventEmitter` also has an `.emit()` method which announces a named event that has occurred.

```
// Require in the 'events' core module
let events = require('events');

// Create an instance of the EventEmitter
class
let myEmitter = new events.EventEmitter();
let version = (data) => {
 console.log(`participant: ${data}.`);
};

// Assign the version function as the
listener callback for 'new user' events
myEmitter.on('new user', version)

// Emit a 'new user' event
myEmitter.emit('new user', 'Lily Pad')
// 'Lily Pad'
```

The error Module

The asynchronous operations involving the Node.js APIs assume that the provided callback functions should have an error passed as the first parameter. If the asynchronous task results in an error, the error will be passed in as the first argument to the callback function. If no error was thrown, then the first argument will be `undefined` .

Input/Output

Input is data that is given to the computer, while output is any data or feedback that a computer provides. In Node, we can get input from a user using the `stdin.on()` method on the `process` object. We are able to use this because `.on()` is an instance of `EventEmitter` . To give an output, we can use the `.stdout.write()` method on the process object as well. This is because `console.log()` is a thin wrapper on `.stdout.write()` .

```
// Recieves an input
process.stdin.on();

// Gives an output
process.stdout.write();
```

The `fs` Module

The *filesystem* controls how data on a computer is stored and retrieved. Node.js provides the `fs` core module, which allows interaction with the filesystem. Each method provided through the module has a synchronous and asynchronous version to allow for flexibility. A method available in the module is the `.readFile()` method that reads data from the provided file.

```
// First argument is the file path
// The second argument is the file's
character encoding
// The third argument is the invoked
function
fs.readFile('./file.txt', 'utf-8',
CallbackFunction);
```

Readable/Writable Streams

In most cases, data isn't processed all at once but rather piece by piece. This is what we call streams. Streaming data is preferred as it doesn't require tons of RAM and doesn't need to have all the data on hand to begin processing it. To read files line-by-line, we can use the `.createInterface()` method from the readline core module. We can write to streams by using the `.createWriteStream()` method.

```
// Readable stream
readline.createInterface();

// Writtable Stream
fs.createWriteStream();
```

The `Buffer` Object

A `Buffer` is an object that represents a static amount of memory that can't be resized. The `Buffer` class is within the global `buffer` module, meaning it can be used without the `require()` statement.

The `.alloc()` Method

The `Buffer` object has the `.alloc()` method that allows a new `Buffer` object to be created with the size specified as the first argument. Optionally, a second argument can be provided to specify the fill and a third argument to specify the encoding.

```
const bufferAlloc = Buffer.alloc(10, 'b');
// Creates a buffer of size 10 filled with
'b'
```

The `.toString()` Method

A `Buffer` object can be translated into a human-readable string by chaining the `.toString()` method to a `Buffer` object. Optionally, encoding can be specified as the first argument, byte offset to begin translating can be provided as the second argument, and the byte offset to end translating as the third argument.

```
const bufferAlloc = Buffer.alloc(5, 'b');
console.log(bufferAlloc.toString()); //
Ouptut: bbbbb
```

The `.from()` Method

A new `Buffer` object can be created from a specified string, array, or another `Buffer` object using the `.from()` method. Encoding can be specified optionally as the second argument.

```
const bufferFrom = Buffer.from('Hello World'); // Creates buffer from 'Hello World' string
```

The `.concat()` Method

The `.concat()` method joins all `Buffer` objects in the specified array into one `Buffer` object. The length of the concatenated `Buffer` can be optionally provided as the second argument. This method is useful because a `Buffer` object can't be resized.

```
const buffer1 = Buffer.from('Hello');
const buffer2 = Buffer.from('World');
const bufferArray = [buffer1, buffer2];

const combinedBuffer
= Buffer.concat(bufferArray);
console.log(combinedBuffer.toString()); //
Logs 'HelloWorld'
```

The `timers` Module

The global `timers` module contains scheduling functions such as `setTimeout()`, `setInterval()`, and `setImmediate()`. These functions are put into a queue processed at every iteration of the Node.js event loop.

The `setImmediate()` Function

The `setImmediate()` function executes the specified callback function after the current event loop has completed. The function accepts a callback function as its first argument and optionally accepts arguments for the callback function as the subsequent arguments.

```
setImmediate(() => {
  console.log('End of this event loop!');
})
```