Advanced Node js

What will we cover

- Modules
- Events
- Stream
- File System
- Error Handling

Module

- Node.js has a simple module loading system. In Node.js, files and modules are in one-to-one correspondence (each file is treated as a separate module).

Core Module

- Node.js has several modules compiled into the binary.
- The core modules are defined within Node.js's source.

Example

- As an example, consider a file named foo.js:

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

- On the first line, foo.js loads the module circle.js that is in the same directory as foo.js.
- Here are the contents of circle.js:
- const PI = Math.PI;

```
module.exports = {
area: function (r) { return PI * r * r ; };
perimeter: function(r) { return 2 * PI * r ; };
}
```

Understanding module.exports and exports in Node.js

- ```
  // using exports
  const PI = Math.PI;
  exports.area : function (r) { return PI * r * r ; };

  exports.perimeter : function(r) { return 2 * PI * r ; };

  }
  ```

- ```
  // using module.exports
  const PI = Math.PI;
  module.exports = {

  area: function (r) { return PI * r * r ; };

  perimeter: function(r) { return 2 * PI * r ; };

  }
  ```

File Modules

- A required module prefixed with '/' is an absolute path to the file. For example, require('/home/marco/foo.js') will load the file at /home/marco/foo.js.

- A required module prefixed with './' is relative to the file calling require(). That is, circle.js must be in the same directory as foo.js for require('./circle') to find it.

- Without a leading '/', './', or '../' to indicate a file, the module must either be a core module or is loaded from a node_modulesfolder.

- If the given path does not exist, require() will throw an Error with its code property set to 'MODULE_NOT_FOUND'.

Caching in Modules

- Modules are cached after the first time they are loaded. This means that every call to `require('foo')` will get exactly the same object returned, if it would resolve to the same file.

- Multiple calls to `require('foo')` may not cause the module code to be executed multiple times.

- If you want to have a module execute code multiple times, then export a function as class, and instantiate that function on each require.

Events

- Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") periodically emit named events that cause Function objects ("listeners") to be called.
- For instance: a net.Server object emits an event each time a peer connects to it; a fs.ReadStream emits an event when the file is opened; a stream emits an event whenever data is available to be read.
- All objects that emit events are instances of the `EventEmitter` class. These objects expose an `eventEmitter.on()` function that allows one or more functions to be attached to named events emitted by the object. Typically, event names are camel-cased strings but any valid JavaScript property key can be used.
- When the `EventEmitter` object emits an event, all of the functions attached to that specific event are called *synchronously*. Any values returned by the called listeners are *ignored* and will be discarded.

Example

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

const myEmitter = new events.EventEmitter();
myEmitter.on('eventwer', () => {
  console.log('an event occurred!');
});
myEmitter.emit('eventwer');
```

Passing arguments and this to listeners

The `eventEmitter.emit()` method allows an arbitrary set of arguments to be passed to the listener functions. It is important to keep in mind that when an ordinary listener function is called by the `EventEmitter`, the standard `this` keyword is intentionally set to reference the `EventEmitter` to which the listener is attached.

```
const myEmitter = new events.EventEmitter();
myEmitter.on('event', function(a, b) {
  console.log(a, b, this);
  // Prints:
  //   a b MyEmitter {
  //     domain: null,
  //     _events: { event: [Function] },
  //     _eventsCount: 1,
  //     _maxListeners: undefined }
});
myEmitter.emit('event', 'a', 'b');
```

Error events

- When an error occurs within an `EventEmitter` instance, the typical action is for an `'error'` event to be emitted. These are treated as special cases within Node.js.

- If an `EventEmitter` does *not* have at least one listener registered for the `'error'` event, and an `'error'` event is emitted, the error is thrown, a stack trace is printed, and the Node.js process exits.

- As a best practice, listeners should always be added for the `'error'` events.

Streams

- Streams are objects that let you read data from a source or write data to a destination in continuous fashion.

- There are many stream objects provided by Node.js. For instance, a request to an HTTP server and process.stdout are both stream instances.

- Streams can be readable, writable, or both. All streams are instances of EventEmitter.

Types of Streams

There are four fundamental stream types within Node.js:

- Readable - streams from which data can be read (for example fs.createReadStream()).
- Writable - streams to which data can be written (for example fs.createWriteStream()).
- Duplex - streams that are both Readable and Writable (for example net.Socket).
- Transform - Duplex streams that can modify or transform the data as it is written and read .

Example

```
var fs = require('fs');
var readableStream = fs.createReadStream('file.txt');
var data = '';
var chunk;

readableStream.on('readable', function() {
    while ((chunk=readableStream.read()) != null) {
        data += chunk;
    }
});

readableStream.on('end', function() {
    console.log(data)
});
```

Piping

Piping is a great mechanism in which you can read data from the source and write to destination without managing the flow yourself. Take a look at the following snippet:

```
var fs = require('fs');
var readableStream = fs.createReadStream('file1.txt');
var writableStream = fs.createWriteStream('file2.txt');

readableStream.pipe(writableStream);
```

File System

- File I/O is provided by simple wrappers around standard POSIX functions. To use this module do `require('fs')`. All the methods have asynchronous and synchronous forms.

- The asynchronous form always takes a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be `null` or `undefined`.

Example

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

// Asynchronous read
fs.readFile('input.txt', function (err, data) {
   if (err) {
      return console.error(err);
   }
   console.log("Asynchronous read: " + data.toString());
});

// Synchronous read
var data = fs.readFileSync('input.txt');
console.log("Synchronous read: " + data.toString());

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

Some useful functions

- fs.access(path[, mode], callback)
- fs.accessSync(path[, mode])
- fs.close(fd, callback)
- fs.closeSync(fd)
- fs.createReadStream(path[, options])
- fs.createWriteStream(path[, options])
- fs.exists(path, callback)
- fs.existsSync(path)`
- fs.readFile(file[, options], callback)
- fs.readFileSync(file[, options])
- fs.readSync(fd, buffer, offset, length, position)
- fs.realpath(path[, options], callback)
- fs.realpathSync(path[, options])
- fs.rename(oldPath, newPath, callback)
- fs.renameSync(oldPath, newPath)
- fs.rmdir(path, callback)
- fs.rmdirSync(path)
- fs.unwatchFile(filename[, listener])
- fs.watchFile(filename[, options], listener)

Errors

Applications running in Node.js will generally experience four categories of errors:

- Standard JavaScript errors such as:
  - <SyntaxError> : thrown in response to improper JavaScript language syntax.
  - <RangeError> : thrown when a value is not within an expected range
  - <ReferenceError> : thrown when using undefined variables
  - <TypeError> : thrown when passing arguments of the wrong type
- System errors triggered by underlying operating system constraints such as attempting to open a file that does not exist, attempting to send data over a closed socket, etc;
- And User-specified errors triggered by application code.
- Assertion Errors are a special class of error that can be triggered whenever Node.js detects an exceptional logic violation that should never occur. These are raised typically by the `assert` module.

Node js style error handling in callbacks

Most asynchronous methods exposed by the Node.js core API follow an idiomatic pattern referred to as a "Node.js style callback". With this pattern, a callback function is passed to the method as an argument. When the operation either completes or an error is raised, the callback function is called with the Error object (if any) passed as the first argument. If no error was raised, the first argument will be passed as null.

```
const fs = require('fs');
function nodeStyleCallback(err, data) {
  if (err) {
    console.error('There was an error', err);
    return;
  }
  console.log(data);
}

fs.readFile('/some/file/that/does-not-exist', nodeStyleCallback);
fs.readFile('/some/file/that/does-exist', nodeStyleCallback);
```

Contd.

The JavaScript try / catch mechanism cannot be used to intercept errors generated by asynchronous APIs. A common mistake for beginners is to try to use throw inside a Node.js style callback:

```
// THIS WILL NOT WORK:
const fs = require('fs');
try {
  fs.readFile('/some/file/that/does-not-exist', (err, data) => {
    // mistaken assumption: throwing here...
    if (err) {
      throw err;
    }
  });
} catch (err) {
  console.error(err);
}
```

Exercise:

1. Print all the files in a folder using node Js.
2. Read data from 1 file and write to another file using streams
3. Write a script to take youtube URL as parameter and download the video (use ‘youtube-dl’ module).
4. Write a file upload API and print the upload process on console

Thanks

Queries

Doubts

Discussion