Node.js Application Development (LFW211)

Lab Exercises (9/5/2022)

Table of Contents

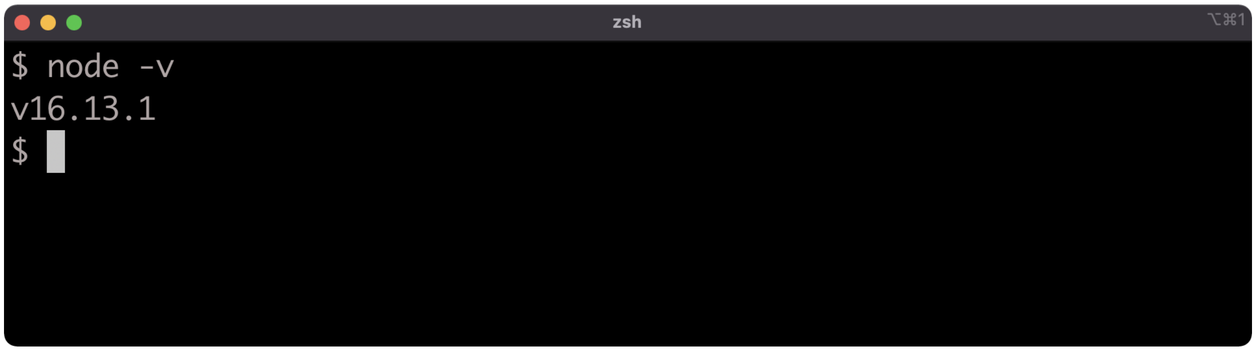
THE LINUX FOUNDATION | Training & Certification

Lab 2.1 - Node Check

To be sure that Node.js is installed and it is right version for this course, run the following command in the terminal:

```
node -v
```

This should output something like the following:



```
$ node -v
v16.13.1
$
```

As long as the first number (followed by "v") is 16, then the correct version of Node is installed for this course.

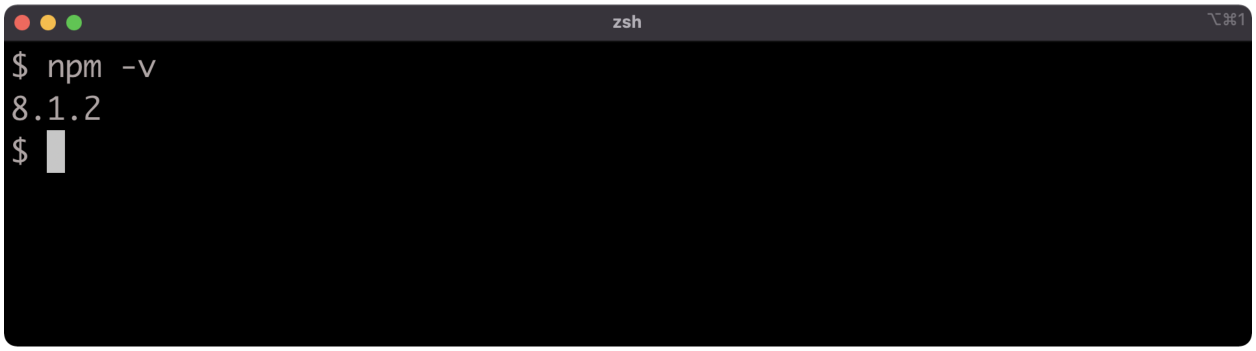
Lab 2.2 - NPM Check

When Node.js is installed, the default package manager for Node is also installed.

To check run:

```
npm -v
```

This should output something like the following:



```
$ npm -v
8.1.2
$ 
```

If the first number before the period is 6 or greater, then we are ready to proceed.

Lab 3.1 - Stack Size

In the labs-1 folder there is a file called `will-throw.js`. Run the file without any flags, and then run the file with `--stack-trace-limit` set to 200.

In the first case, there should only be ten stack frames in the error output.

In the second case, there should be more than ten frames in the error in output.

Lab 3.2 - Checking Syntax

In the labs-2 folder there are two files, `bad-syntax.js` and `correct-syntax.js`, use the appropriate flag to check the syntax of each file.

There should be no output when checking the syntax of `correct-syntax.js`.

There should be a Syntax Error when checking the syntax of `bad-syntax.js`.

THE LINUX FOUNDATION | Training & Certification

Lab 4.1 - Break Before User Code Starts

The labs-1 folder contains an `app.js` file. Start `app.js` with Node in Inspect Mode, but with the application immediately paused on the first line of executable code.

If this task is performed correctly, the program should be paused on line 5. This can be verified by connecting Chrome Devtools to the Inspector via the "chrome://inspect" URI, finding the process under "Remote Targets", clicking "inspect" and ensuring the "Sources" tab is selected.

Lab 4.2 - Setting a Breakpoint Using a Code Keyword

The labs-2 folder has an **`app.js`** file, edit the file so that when started in Inspect mode it will pause in the first line of the **`f`** function.

To verify, start the **`app.js`** file in Inspect mode (without breaking before user code starts). Connect Chrome Devtools, and ensure the "Sources" tab is selected. All being well, the application should be paused at line 2.

Lab 5.1 - Closure Scope

In the labs-1 folder an `app.js` file contains the following:

```
'use strict'
const sayHiTo = prefixer('Hello ')
const sayByeTo = prefixer('Goodbye ')
console.log(sayHiTo('Dave')) // prints 'Hello Dave'
console.log(sayHiTo('Annie')) // prints 'Hello Annie'
console.log(sayByeTo('Dave')) // prints 'Goodbye Dave'
```

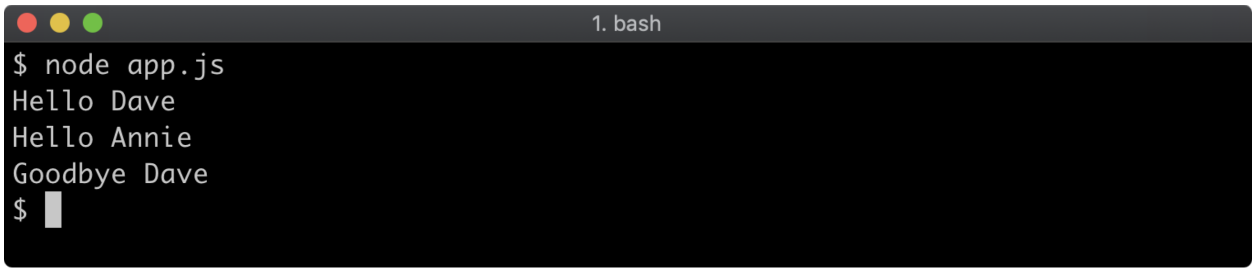
Implement the `prefixer` function.

When ready, open a terminal, navigate to the labs-1 folder and run your program with:

```
node app.js
```

This program will not run correctly until the implementation is complete.

When successfully implemented the output should be as follows:



```
$ node app.js
Hello Dave
Hello Annie
Goodbye Dave
$ 
```

Lab 5.2 - Prototypal Inheritance

The labs-2 folder contains an **app.js** file with the following:

```
const assert = require('assert')

// TODO:
// implement a way to create a prototype chain
// of leopard -> lynx -> cat
// leopard prototype must have ONLY a hiss method
// lynx prototype must have ONLY a purr method
// cat prototype must have ONLY a meow method

const felix = null //TODO replace null with instantiation of a cat
felix.meow() // prints Felix the cat: meow
felix.purr() // prints Felix the cat: prrr
felix.hiss() // prints Felix the cat: hsss

// prototype checks, do not remove
const felixProto = Object.getPrototypeOf(felix)
const felixProtoProto = Object.getPrototypeOf(felixProto)
const felixProtoProtoProto = Object.getPrototypeOf(felixProtoProto)

assert(Object.getOwnPropertyNames(felixProto).length, 1)
assert(Object.getOwnPropertyNames(felixProtoProto).length, 1)
assert(Object.getOwnPropertyNames(felixProtoProto).length, 1)
assert(typeof felixProto.meow, 'function')
assert(typeof felixProtoProto.purr, 'function')
assert(typeof felixProtoProtoProto.hiss, 'function')
```

```
console.log('prototype checks passed!')
```

Using any of the approaches described create a prototypal inheritance chain and then create an instance from that chain, and assign it to **`felix`** such that:

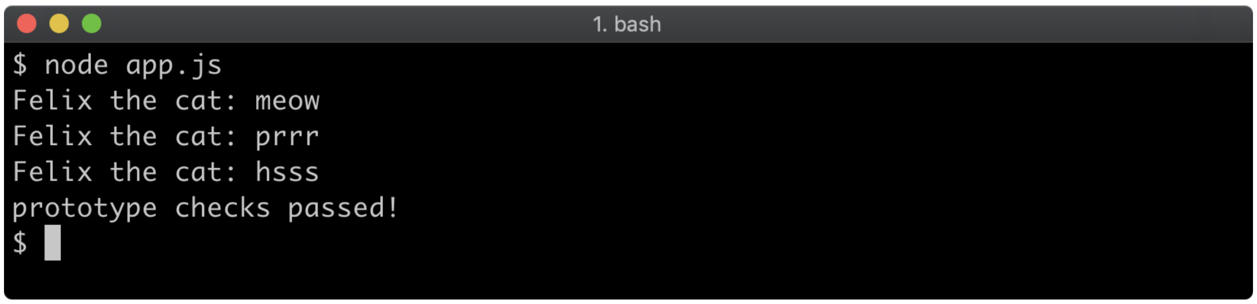
- The prototype of **`felix`** should be an object with a **`meow`** method
- The prototype of that object should be an object with a **`purr`** method
- The prototype of that object should be an object with a **`hiss`** method

When ready, open a terminal, navigate to the labs-2 folder and run your implementation with

```
node app.js
```

This program will not run correctly, it will throw exceptions, until the implementation is complete.

If correctly implemented **`app.js`** should output: `prototype checks passed!`:



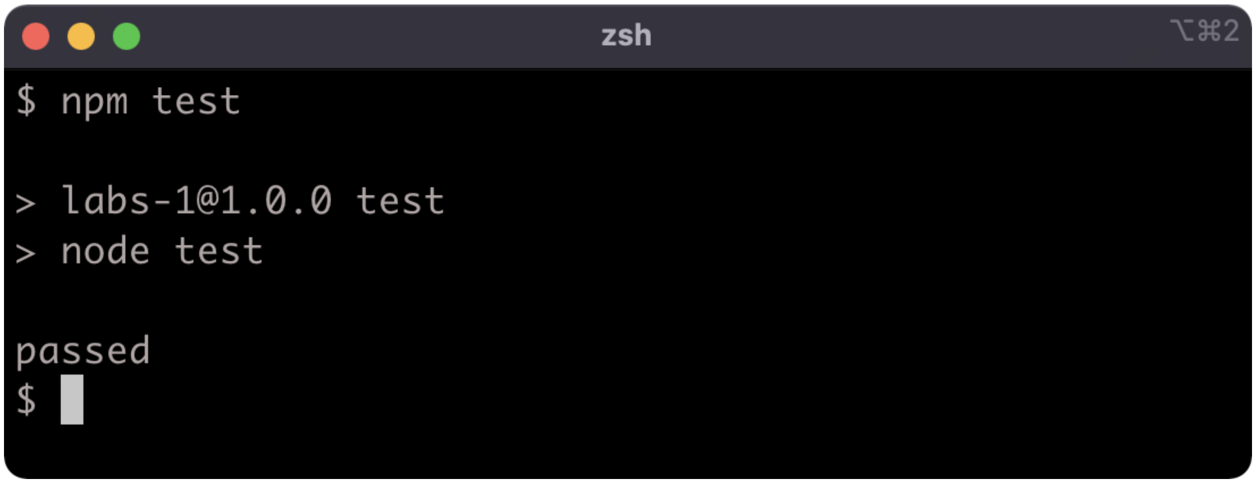
```
$ node app.js
Felix the cat: meow
Felix the cat: prrr
Felix the cat: hsss
prototype checks passed!
$ 
```

It's not necessary for **`meow`**, **`purr`** and **`hiss`** methods **`console.log`** exactly the same output, or even to output anything at all. Making Felix meow, purr and hiss is a nice-to-have. The most important thing is that the prototype chain is correctly set up.

Lab 6.1 - Install a Development Dependency

The labs-1 folder has a **`package.json`** file in it. Install **`nonsynchronous`** (https://www.npmjs.com/package/nonsynchronous) as a development dependency.

Run **`npm test`** in the labs-1 folder to check that the task has been completed:



```
$ npm test

> labs-1@1.0.0 test
> node test

passed
$
```

If the output says "passed" then the task was completed correctly.

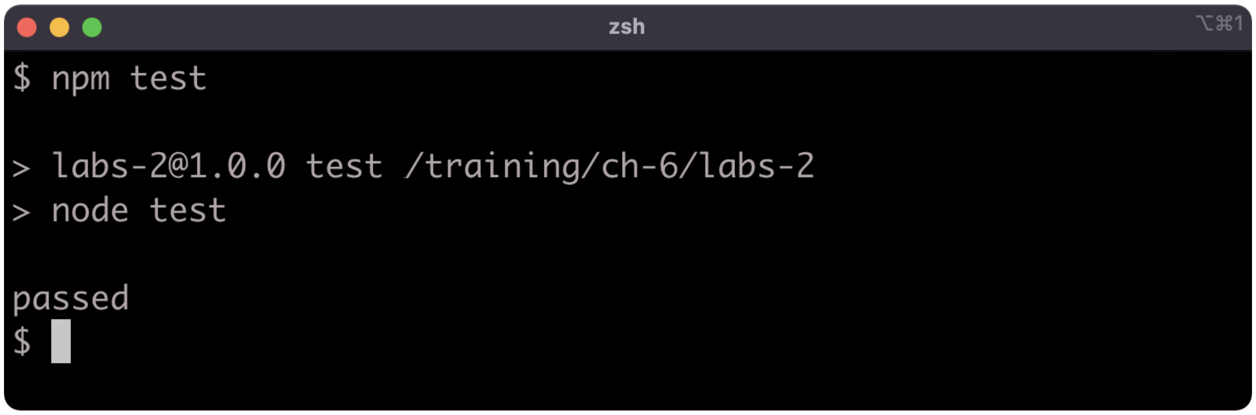
Lab 6.2 - Install a Dependency Using a Semver Range

The labs-2 folder contains a `package.json` file.

Install the following dependencies at the specified version ranges, and ensure that those ranges are correctly specified in the `package.json` file:

- Install **`fastify`** at greater than or equal to 2.0.0, while accepting all future MINOR and PATCH versions
- Install **`rfdc`** at exactly version 1.1.3

Run **`npm install`** to install the development dependency required to validate this exercise, and then run **`npm test`** in the labs-2 folder to check that the task has been completed:



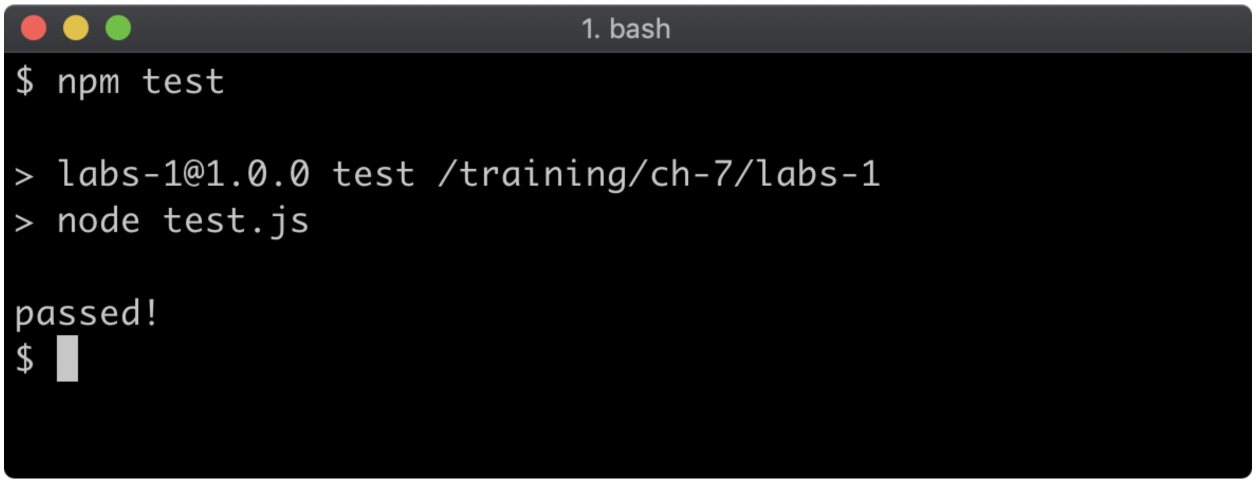
```
$ npm test

> labs-2@1.0.0 test /training/ch-6/labs-2
> node test

passed
$
```

If the output says "passed" then the task was completed correctly.

Lab 7.1 - Creating a Module

The labs-1 folder has an `index.js` file. Write a function that takes two numbers and adds them together, and then export that function from the `index.js` file.

Run `npm test` to check whether `index.js` was correctly implemented. If it was the output should contain "`passed!`":



```
$ npm test

> labs-1@1.0.0 test /training/ch-7/labs-1
> node test.js

passed!
$
```

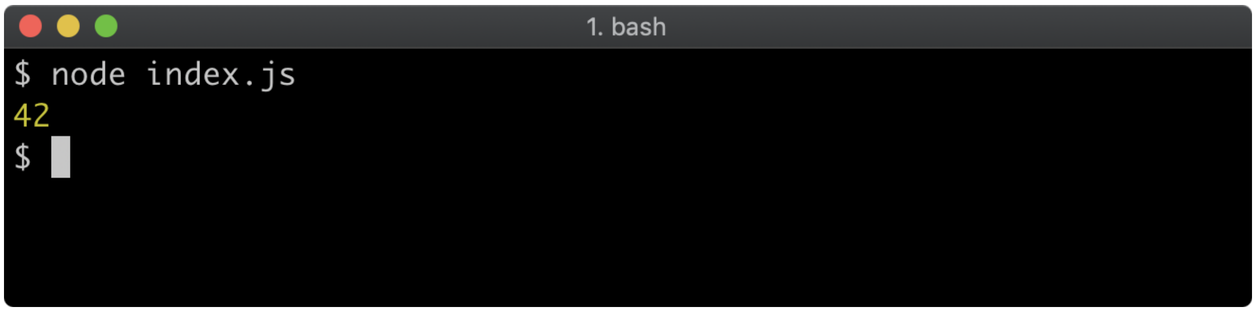
By default, the labs-1 folder is set up as a CJS project, but if desired, the `package.json` can be modified to convert to an ESM module (by either setting the `type` field to `module` or renaming `index.js` to `index.mjs` and setting the `type` field accordingly). The exercise can be completed either with the default CJS or with ESM or both.

Lab 7.2 - Loading a Module

The labs-2 is a sibling to labs-1.In the `index.js` file of labs-2, load the module that was created in the previous lab task and use that module to `console.log` the sum of 19 and 23.

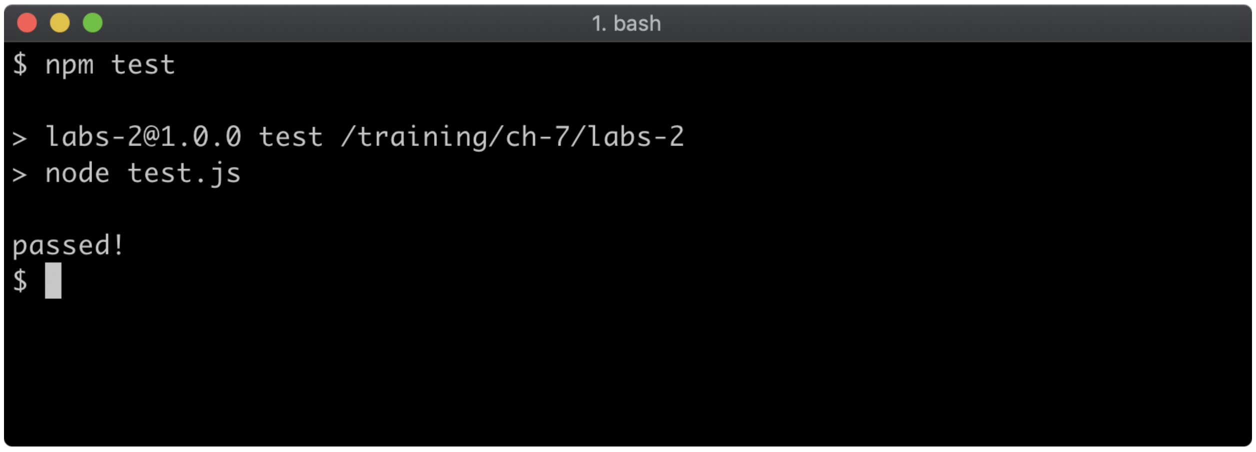
The labs-2 folder is set up as a CJS project. Recall that ESM can load CJS but CJS cannot load ESM during initialization. If the prior exercise was completed as an ESM module it *cannot* be synchronously loaded into a CJS module. Therefore if the prior exercise was completed in the form of an ESM module, this exercise must also be similarly converted to ESM.

When `index.js` is executed with `node` it should output 42:



```
$ node index.js
42
$
```

Run `npm test` to check whether `index.js` was correctly implemented. If it was, the output should contain "`passed!`":



```
$ npm test

> labs-2@1.0.0 test /training/ch-7/labs-2
> node test.js

passed!
$
```

Lab 8.1 - Parallel Execution

In the labs-1 folder, the **parallel.js** file contains the following:

```
const print = (err, contents) => {
  if (err) console.error(err)
  else console.log(contents)
}

const opA = (cb) => {
  setTimeout(() => {
    cb(null, 'A')
  }, 500)
}

const opB = (cb) => {
  setTimeout(() => {
    cb(null, 'B')
  }, 250)
}

const opC = (cb) => {
  setTimeout(() => {
    cb(null, 'C')
  }, 125)
}
```

The **opA** function must be called before **opB**, and **opB** must be called before **opC**.

Call functions in `parallel.js` in such a way that `C` then `B` then `A` is printed out.

Lab 8.2 - Serial Execution

In the labs-2 folder, the `serial.js` file contains the following:

```
const { promisify } = require('util')

const print = (err, contents) => {
  if (err) console.error(err)
  else console.log(contents)
}

const opA = (cb) => {
  setTimeout(() => {
    cb(null, 'A')
  }, 500)
}

const opB = (cb) => {
  setTimeout(() => {
    cb(null, 'B')
  }, 250)
}

const opC = (cb) => {
  setTimeout(() => {
    cb(null, 'C')
  }, 125)
}
```

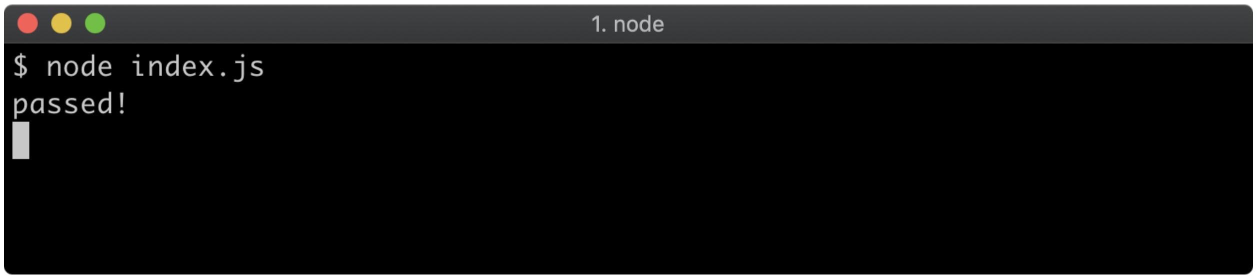
Call the functions in such a way that `A` then `B` then `C` is printed out.
Remember **`promisify`** can be used to convert a callback API to a promise-based API.
The **`promisify`** function is included at the top of **`serial.js`** in case a promise based solution is prefered.

Lab 9.1 - Single Use Listener

The labs-1 `index.js` file contains the following code:

```
'use strict'
const assert = require('assert')
const { EventEmitter } = require('events')

const ee = new EventEmitter()
let count = 0
setInterval(() => {
  ee.emit('tick')
}, 100)

function listener () {
  count++
  setTimeout(() => {
    assert.equal(count, 1)
    assert.equal(this, ee)
    console.log('passed!')
  }, 250)
}
```

Register the **`listener`** function with the **`ee`** event emitter in such a way that the **`listener`** function is only called a single time. If implemented correctly, the program should print out `passed!`:



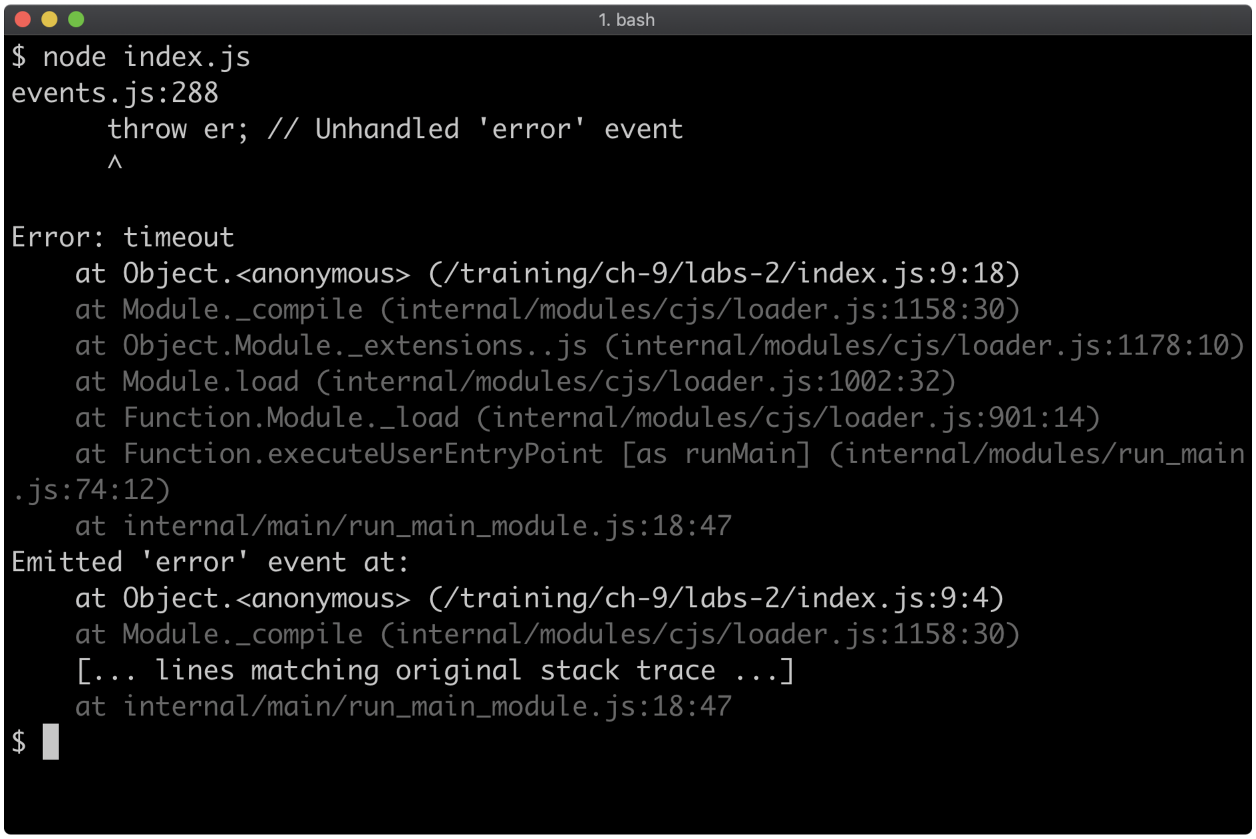
```
$ node index.js
passed!
```

Lab 9.2 - Implementing a Timeout Error

The labs-2 folder has an `index.js` file containing the following:

```
'use strict'
const { EventEmitter } = require('events')

process.nextTick(console.log, 'passed!')

const ee = new EventEmitter()

ee.emit('error', Error('timeout'))
```

Currently the process crashes:



```
$ node index.js
events.js:288
      throw er; // Unhandled 'error' event
      ^

Error: timeout
    at Object.<anonymous> (/training/ch-9/labs-2/index.js:9:18)
    at Module._compile (internal/modules/cjs/loader.js:1158:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1178:10)
    at Module.load (internal/modules/cjs/loader.js:1002:32)
    at Function.Module._load (internal/modules/cjs/loader.js:901:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main
.js:74:12)
    at internal/main/run_main_module.js:18:47
Emitted 'error' event at:
    at Object.<anonymous> (/training/ch-9/labs-2/index.js:9:4)
    at Module._compile (internal/modules/cjs/loader.js:1158:30)
    [... lines matching original stack trace ...]
    at internal/main/run_main_module.js:18:47
$ 
```

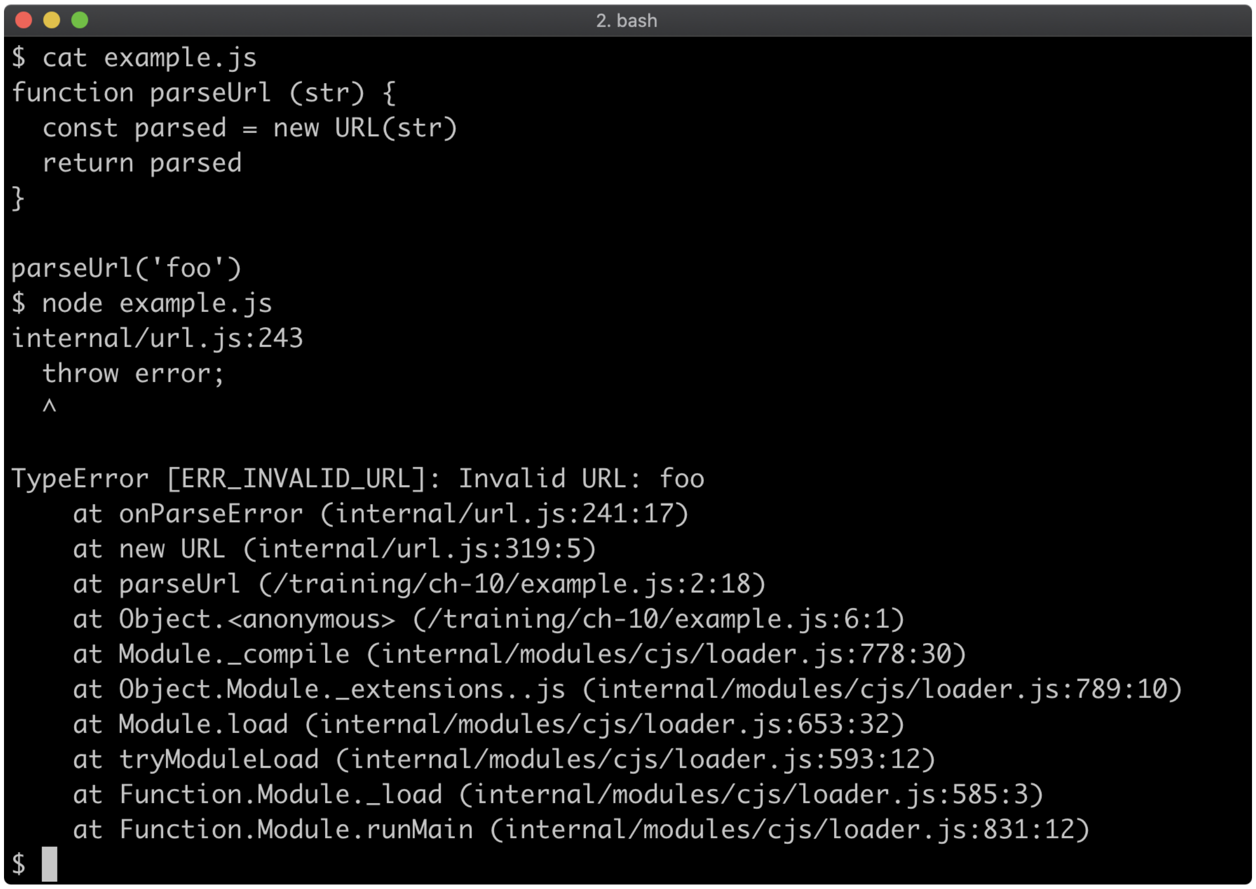
Without removing any of the existing code, and without using a **`try/catch`** block add some code which stops the process from crashing. When implemented correctly the process should output `passed!`.

Lab 10.1 - Synchronous Error Handling

The native **`URL`** constructor can be used to parse URLs, it's been wrapped into a function called **`parseURL`**:

```
function parseURL (str) {
  const parsed = new URL(str)
  return parsed
}
```

If **`URL`** is passed a unparsable URL string it will throw, so calling **`parseURL('foo')`** will result in an exception:



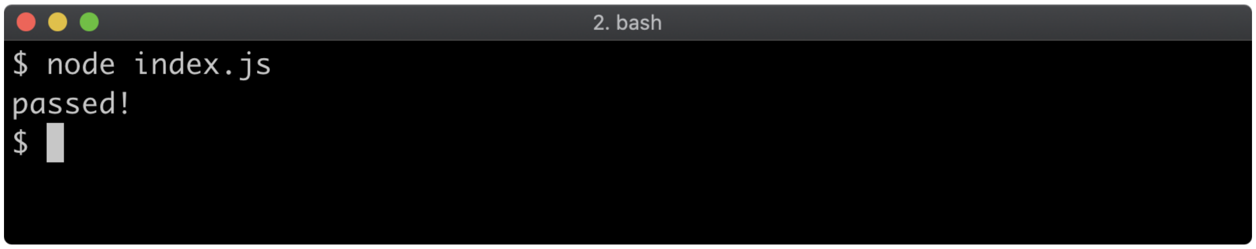
```
$ cat example.js
function parseUrl (str) {
  const parsed = new URL(str)
  return parsed
}

parseUrl('foo')
$ node example.js
internal/url.js:243
  throw error;
  ^

TypeError [ERR_INVALID_URL]: Invalid URL: foo
    at onParseError (internal/url.js:241:17)
    at new URL (internal/url.js:319:5)
    at parseUrl (/training/ch-10/example.js:2:18)
    at Object.<anonymous> (/training/ch-10/example.js:6:1)
    at Module._compile (internal/modules/cjs/loader.js:778:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:789:10)
    at Module.load (internal/modules/cjs/loader.js:653:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
    at Function.Module._load (internal/modules/cjs/loader.js:585:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:831:12)
$
```

The labs-1 folder contains an `index.js` file with the following content:

```
'use strict'
const assert = require('assert')

function parseUrl (str) {
  const parsed = new URL(str)
  return parsed
}

assert.doesNotThrow(() => { parseUrl('invalid-url') })
assert.equal(parseUrl('invalid-url'), null)
assert.deepEqual(
  parseUrl('http://example.com'),
  new URL('http://example.com')
)
console.log('passed!')
```

Modify the `parseURL` function body such that instead of throwing an error, it returns `null` when the URL is invalid. Use the fact that `URL` will throw when given invalid input to determine whether or not to return `null` or a parsed object.

Once implemented, execute the `index.js` file with node, if the output says `passed!` then the exercise was completed successfully:



```
$ node index.js
passed!
$
```

THE LINUX FOUNDATION | Training & Certification

Lab 10.2 - Async Function Error Handling

The following code loads the **`fs`** module and uses its promises interface to read a file based on a file path passed to a **`read`** function:

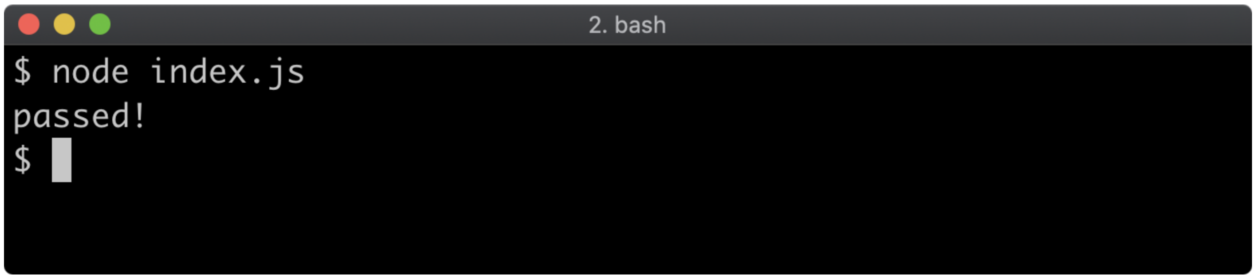
```
const fs = require('fs')

async function read (file) {
  const content = await fs.promises.readFile(file)
  return content
}
```

The promise returned from **`fs.promises.readFile`** may reject for a variety of reasons, for instance if the specified file path doesn't exist or the process doesn't have permissions to access it. In this scenario, we don't care what the reason for failure is, we just want to propagate a single error instance from the native **`Error`** constructor with the message 'failed to read'.

The labs-2 **`index.js`** file contains the following code:

```
'use strict'
const fs = require('fs')
const assert = require('assert')

async function read (file) {
  const content = await fs.promises.readFile(file)
  return content
}

async function check () {
  await assert.rejects(
    read('not-a-valid-filepath'),
    new Error('failed to read')
  )
  assert.deepEqual(
    await read(__filename),
    fs.readFileSync(__filename)
  )
  console.log('passed!')
}

check()
```

Modify the body of the **`read`** function so that any possible rejection by the promise returned from the **`fs.promises.readFile`** call results in the **`read`** function rejecting with a **`new Error('failed to read')`** error. If implemented correctly, when **`node index.js`** is executed the output should be `passed!`:



```
$ node index.js
passed!
$ 
```

Lab 11.1 - Create a Buffer Safely

The `index.js` file in the lab-1 folder contains the following:

```
'use strict'
const assert = require('assert')
const buffer = Buffer.allocUnsafe(4096)
console.log(buffer)

for (const byte of buffer) assert.equal(byte, 0)
console.log('passed!')
```

Alter the code so that the `buffer` is safely allocated. Do not explicitly fill the buffer with anything. If the process prints the buffer and then logs `passed!`, the exercise was correctly completed.

Lab 11.2 - Convert a String to base64 Encoded String by Using a Buffer

The labs-2 **`index.js`** has the following code:

```
'use strict'
const assert = require('assert')
const str = 'buffers are neat'
const base64 = '' // convert str to base64

console.log(base64)

assert.equal(base64, Buffer.from([
  89,110,86,109,90,109,86,121,99,
  121,66,104,99,109,85,103,98,109,
  86,104,100,65,61,61]))

console.log('passed!')
```

Using the **`Buffer`** API in some way, edit the code so that the **`base64`** constant contains a base64 representation of the **`str`** constant.

If the process prints the base64 string and then logs **`passed!`**, the exercise was correctly completed.

Lab 12.1 - Piping

The labs-1 folder has an `index.js` file containing the following:

```
'use strict'
const { Readable, Writable } = require('stream')
const assert = require('assert')
const createWritable = () => {
  const sink = []
  let piped = false
  setImmediate(() => {
    assert.strictEqual(piped, true, 'use the pipe method')
    assert.deepStrictEqual(sink, ['a', 'b', 'c'])
  })
  const writable = new Writable({
    decodeStrings: false,
    write(chunk, enc, cb) {
      sink.push(chunk)
      cb()
    },
    final() {
      console.log('passed!')
    }
  })
  writable.once('pipe', () => {
    piped = true
  })
  return writable
}
```

```
const readable = Readable.from(['a', 'b', 'c'])
const writable = createWritable()

// TODO - send all data from readable to writable:
```

Use the appropriate method to make sure that all data in the `readable` stream is automatically sent to the `writable` stream.

If successfully implemented the process will output: `passed!`

Lab 12.2 - Create a Transform Stream

The labs-2 folder has an `index.js` file containing the following:

```
'use strict'
const { Readable, Writable, Transform, PassThrough, pipeline } =
require('stream')
const assert = require('assert')
const createWritable = () => {
  const sink = []
  const writable = new Writable({
    write(chunk, enc, cb) {
      sink.push(chunk.toString())
      cb()
    }
  })
  writable.sink = sink
  return writable
}
const readable = Readable.from(['a', 'b', 'c'])
const writable = createWritable()

// TODO: replace the pass through stream
// with a transform stream that uppercases
// incoming characters
const transform = new PassThrough()

pipeline(readable, transform, writable, (err) => {
  assert.ifError(err)
  assert.deepStrictEqual(writable.sink, ['A', 'B', 'C'])
  console.log('passed!')
})
```

Replace the line that states `const transform = new PassThrough()` so that `transform` is assigned to a transform stream that upper cases any incoming chunks. If successfully implemented the process will output: `passed!`

Lab 13.1 - Read Directory and Write File

The labs-1 folder contains an `index.js` file containing the following:

```
'use strict'
const assert = require('assert')
const { join } = require('path')
const fs = require('fs')
const project = join(__dirname, 'project')
try { fs.rmdirSync(project, {recursive: true}) } catch (err) {}
const files = Array.from(Array(5), () => {
  return join(project, Math.random().toString(36).slice(2))
})
fs.mkdirSync(project)
for (const f of files) fs.closeSync(fs.openSync(f, 'w'))

const out = join(__dirname, 'out.txt')

function exercise () {
  // TODO read the files in the project folder
  // and write them to the out.txt file
}

exercise()
assert(fs.readFileSync(out).toString(), files.toString())
console.log('passed!')
```

The above code will generate a project folder and add five files to it with pseudo-randomly generated filenames.

Complete the function named **`exercise`** so that all the files in the **`project`** folder, as stored in the **`project`** constant, are written to the **`out.txt`** file as stored in the **`out`** constant. Only the file names should be stored, not the full file paths, and file names should be comma separated.

For instance, given a **`project`** folder with the following files:

- **`0p2ly0dluiw`**
- **`2ftl32u5zu5`**
- **`8t4iilscua6`**
- **`90370mamnse`**
- **`zfw8w7f8sm8`**

The **`out.txt`** should then contain:

```
0p2ly0dluiw,2ftl32u5zu5,8t4iilscua6,90370mamnse,zfw8w7f8sm8
```

If successfully implemented, the process will output: **`passed!`**.

Lab 13.2 - Watching

The labs-2 folder contains an `index.js` file with the following:

```
'use strict'
const assert = require('assert')
const { join } = require('path')
const fs = require('fs')
const { promisify } = require('util')
const timeout = promisify(setTimeout)
const project = join(__dirname, 'project')
try { fs.rmdirSync(project, {recursive: true}) } catch (err) {
  console.error(err)
}
fs.mkdirSync(project)

let answer = ''

async function writer () {
  const { open, chmod, mkdir } = fs.promises
  const pre = join(project, Math.random().toString(36).slice(2))
  const handle = await open(pre, 'w')
  await handle.close()
  await timeout(500)
  exercise(project)
  const file = join(project, Math.random().toString(36).slice(2))
  const dir = join(project, Math.random().toString(36).slice(2))
  const add = await open(file, 'w')
  await add.close()
  await mkdir(dir)
  await chmod(pre, 0o444)
  await timeout(500)
  assert.strictEqual(
    answer,
    file,
    'answer should be the file (not folder) which was added'
  )
  console.log('passed!')
  process.exit()
}

writer().catch((err) => {
  console.error(err)
  process.exit(1)
})


function exercise (project) {
  const files = new Set(fs.readdirSync(project))
  fs.watch(project, (evt, filename) => {
    try {
      const filepath = join(project, filename)
      const stat = fs.statSync(filepath)

      // TODO - only set the answer variable if the filepath
      // is both newly created AND does not point to a directory

      answer = filepath
    } catch (err) {

    }
  })
}
```

When executed (e.g. using **`node index.js`**) this code will create a folder named **`project`** (removing it first if it already exists and then recreating it), and then perform some file system manipulations within the **`project`** folder.

The **`writer`** function will create a file before calling the **`exercise`** function, to simulate a pre-existing file, The **`exercise`** function will then be called which sets up a file watcher with **`fs.watch`**. The **`writer`** function then proceeds to create a file, a directory and changes the permissions of the previously existing file. These changes will trigger the listener function passed as the second argument to **`fs.watch`**.

The goal is to ensure that the **`answer`** variable is set to the newly created file. So when a directory is added, the **`answer`** variable should not be set to the directory path. When the preexisting files status is updated via a permissions change, the **`answer`** variable should not be set to that preexisting file.

If implemented correctly the process will output: **`passed!`**.

Lab 14.1 - Identifying OS and Exiting

The labs-1 folder contains an empty `index.js` file and a `test.js` file.

The `test.js` file contains the following:

```
'use strict'
const { spawnSync } = require('child_process')
const assert = require('assert')
const { status, stdout } = spawnSync(process.argv[0], [__dirname])

assert.notStrictEqual(status, 0, 'must exit with a non-zero code')
assert.match(
  stdout.toString(),
  /^(d|w|l|aix|.+bsd|sunos|gnu)/i,
  'must output OS identifier'
)
console.log('passed!')
```

In `index.js` use `console.log` to output the operating system identifier. Ensure the process exits with a non-zero exit code.

Run `node test.js` to verify whether the task was successfully completed, if it was `node test.js` will output `passed!`.

Lab 14.2 - OS Uptime and Memory

The labs-2 folder contains an `index.js` file and a `test.js` file.

The `index.js` file contains the following:

```
'use strict'
setTimeout(() => {
  console.log() // TODO output uptime of process
  console.log() // TODO output uptime of OS
  console.log() // TODO output total system memory
  console.log() // TODO output total heap memory
}, 1000)
```

Follow the TODO comments for each of the `console.log` statements.

To verify the implementation, the `test.js` file contains the following:

```
'use strict'
const assert = require('assert')
const os = require('os')
const { runInThisContext } = require('vm')
const run = (s) => runInThisContext(Buffer.from(s, 'base64'))
const { log } = console
const queue = [
  (line) => assert.strictEqual(
    Math.floor(line),
    1,
    'first log line should be the uptime of the process'
  ),
  (line) => assert.strictEqual(
    line,
    run('KG9zKSA9PiBvcy51cHRpbWUoKQ==')(os),
    'second log line should be the uptime of the OS'
  ),
  (line) => assert.strictEqual(
    line,
    run('KG9zKSA9PiBvcy50b3RhbG1lbSgp')(os),
    'third line should be total system memory'
  ),
  (line) => assert.strictEqual(
    line,
    run('cHJvY2Vzcy5tZW1vcnlVc2FnZSgpLmhlYXBUb3RhbA=='),
    'fourth line should be total process memory'
  )
]
console.log = (line) => {
  queue.shift()(line)
  if (queue.length === 0) {
    console.log = log
    console.log('passed!')
  }
}
require('.')
```

Run `node test.js` to verify whether the task was successfully completed, if it was `node test.js` will output `passed!`.

Lab 15.1 - Set Child Process Environment Variable

The labs-1 folder contains an `index.js`, a `child.js` file and a `test.js` file.

The `child.js` file contains the following:

```
'use strict'
const assert = require('assert')
const clean = (env) => Object.fromEntries(
  Object.entries(env).filter(([k]) => !/^(_.*|pwd|shlvl)/i.test(k))
)
const env = clean(process.env)

assert.strictEqual(env.MY_ENV_VAR, 'is set')
assert.strictEqual(
  Object.keys(env).length,
  1,
  'child process should have only one env var'
)
console.log('passed!')
```

The code in `child.js` expects that there will be only one environment variable named **`MY_ENV_VAR`** to have the value **`'is set'`**. If this is not the case the **`assert.strictEqual`** method will throw an assertion error. In certain scenarios some extra environment variables are added to child processes, these are stripped so that there should only ever be one environment variable set in `child.js`, which is the **`MY_ENV_VAR`** environment variable.

The `index.js` file has the following contents:

```
'use strict'
const assert = require('assert')

function exercise (myEnvVar) {
  // TODO return a child process with
  // a single environment variable set
  // named MY_ENV_VAR. The MY_ENV_VAR
  // environment variable's value should
  // be the value of the myEnvVar parameter
  // passed to this exercise function
}
```

Using any **`child_process`** method except **`execFile`** and **`execFileSync`**, complete the exercise function so that it returns a child process that executes the `child.js` file with **`node`**.

To check the exercise implementation, run **`node test.js`**, if successful the process will output: `passed!`. If unsuccessful, various assertion error messages will be output to help provide hints.

One very useful hint up front is: use **`process.execPath`** to reference the **`node`** executable instead of just passing **`'node'`** as string to the **`child_process`** method.

The contents of the `test.js` file is esoteric, and the need to understand the code is minimal, however the contents of `test.js` are shown here for completeness:

```
'use strict'
const assert = require('assert')
const { equal } = assert.strict
const exercise = require('.')

let sp = null
try {
  sp = exercise('is set')
  assert(sp, 'exercise function should return a child process
instance')
  if (Buffer.isBuffer(sp)) {
    equal(sp.toString().trim(), 'passed!', 'child process
misconfigured')
    process.stdout.write(sp)
    return
  }
} catch (err) {
  const { status} = err
  if (status == null) throw err
  equal(status, 0, 'exit code should be 0')
  return
}

if (!sp.on) {
  const { stdout, stderr } = sp
  if (stderr.length > 0) process.stderr.write(stderr)
  if (stdout.length > 0) process.stdout.write(stdout)
  equal(sp.status, 0, 'exit code should be 0')
  equal(stdout.toString().trim(), 'passed!', 'child process
misconfigured')
  return
}

let out = ''
if (sp.stderr) sp.stderr.pipe(process.stderr)
if (sp.stdout) {
  sp.stdout.once('data', (data) => { out = data })
  sp.stdout.pipe(process.stdout)
} else {
  // stdio may be misconfigured, or fork method may be used,
  // allow benefit of the doubt since in either case
  // exit code check will still fail:
  out = 'passed!'
}
const timeout = setTimeout(() => {
  equal(out.toString().trim(), 'passed!', 'child process
misconfigured')
}, 1000)

sp.once('exit', (status) => {
  equal(status, 0, 'exit code should be 0')
  equal(out.toString().trim(), 'passed!', 'child process
misconfigured')
  clearTimeout(timeout)
})
```

The `test.js` file allows for alternative approaches, once the `exercise` function has been completed with one `child_process` method, re-attempt the exercise with a different `child_process` method.

Lab 15.2 - STDIO Redirection

The labs-2 folder `index.js` file contains the following:

```
'use strict'

const { spawn } = require('child_process')

function exercise (command, args) {
  return spawn(command, args)
}

module.exports = exercise
```

Complete the **`exercise`** function so that the returned child process:

- Has no ability to read STDIN.
- Redirects its STDOUT to the parent process' STDOUT.
- Exposes STDERR as a readable stream.

The labs-2 folder also contains a `test.js` file.

To verify that the exercise was completed successfully run **`node test.js`**, if the implementation is correct the process will output: `passed!`.

It is unnecessary to understand the contents of the `test.js` file, but the contents of it are as follows:

```
'use strict'
```

```
const exercise = require('.')
const cp = require('child_process')
const assert = require('assert')
const { equal } = assert.strict
const { SCENARIO } = process.env
const [ node ] = process.argv

const stdoutCheck = () => { exercise(node, ['-p', `'test'`]) }
const stderrCheck = () => {
  const sp = exercise(node, ['-e', `console.error('test')`])
  if (sp.stderr) sp.stderr.pipe(process.stderr)
}
const stdinCheck = () => {
  exercise(node, ['-e', `
      process.stdout.write(Buffer.from([0]))
      process.stdin.pipe(process.stdout)
      setTimeout(() => {
        process.stdout.write(Buffer.from([1]))
      }, 100)
  `])
}

function test (scenario = 0) {
  switch (scenario) {
    case 1: return stdoutCheck()
    case 2: return stderrCheck()
    case 3: return stdinCheck()
  }

  const s1 = cp.spawnSync(node, [__filename], {
    env: {SCENARIO: 1},
  })
  equal(s1.stdout.toString().trim(), 'test', 'should inherit stdout')

  const s2 = cp.spawnSync(node, [__filename], {
    env: {SCENARIO: 2},
  })
  equal(s2.stderr.toString().trim(), 'test', 'should expose stderr')

  const s3 = cp.spawnSync(node, [__filename], {
    input: 'some input',
    env: {SCENARIO: 3},
  })
  equal(s3.stdout.length, 2, 'stdin should be ignored')

  console.log('passed!')
}

test(Number(SCENARIO))
```

Lab 16.1 - Test a Sync API

The labs-1 folder contains a `package.json` file and an `uppercase.js` file.

The `package.json` file contains the following:

```
{
  "name": "labs-1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "UNLICENSED"
}
```

The `uppercase.js` file contains the following:

```
'use strict'

function uppercase (str) {
  if (typeof str !== 'string') throw Error('input must be a string')
  return str.toUpperCase()
}

module.exports = uppercase
```

Write tests for the `uppercase.js` file. Ensure that when `npm test` is executed with the labs-1 folder as the current working directory the `uppercase.js` file is fully tested.

Any additional dependencies, such as a test harness, may be additionally installed.

Also in the labs-1 folder is a `validate.js` file. The implementation can be checked with `node validate.js`. The implementation is successful if the final output of `node validate.js` is `passed!`.

For completeness the following is the `validate.js` file contains the following, but it is not necessary to understand it for the purposes of this exercise:

```
'use strict'
const assert = require('assert').strict
const { spawnSync } = require('child_process')
const { writeFileSync } = require('fs')
const uppercase = require.resolve('./uppercase')
const uppercaseCode =
Buffer.from('2775736520737472696374270a66756e6374696f6e207570706572636
17365202873747229207b0a202069662028747970656f662073747220213d3d2027737
472696e672729207468726f77204572726f722827696e707574206d757374206265206
120737472696e6727290a202072657475726e207374722e746f5570706572436173652
8290a7d0a0a6d6f64756c652e6578706f727473203d20757070657263617365',
'hex')

try {
  {
    writeFileSync(uppercase, uppercaseCode)

    const sp = spawnSync('npm', ['test'], {
      stdio: 'ignore'
    })

    assert.equal(sp.status, 0, 'tests should be successful (is
package.json test script configured?)')
  }

  {
    const badOutput = `'use strict'
    function uppercase (str) {
      if (typeof str !== 'string') throw Error('input must be a
string')
```

```
      return 'bad output'
    }

    module.exports = uppercase
    `

    writeFileSync(uppercase, badOutput)

    const sp = spawnSync('npm', ['test'], {
      stdio: 'ignore'
    })

    assert.equal(sp.status, 1, 'output should be tested')
  }

  {
    const badValidation = `'use strict'
    function uppercase (str) {
      return str.toUpperCase()
    }

    module.exports = uppercase
    `

    writeFileSync(uppercase, badValidation)

    const sp = spawnSync('npm', ['test'], {
      stdio: 'ignore'
    })

    assert.equal(sp.status, 1, 'error case should be tested')
  }
  console.log('passed!')
} finally {
  writeFileSync(uppercase, uppercaseCode)
}
```

Lab 16.2 - Test a Callback-Based API

The labs-2 folder contains a `package.json` file and a `store.js` file.

The `package.json` file contains the following:

```
{
  "name": "labs-2",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "UNLICENSED"
}
```

The `store.js` file contains the following:

```
'use strict'
module.exports = (value, cb) => {
  if (Buffer.isBuffer(value) === false) {
    cb(Error('input must be a buffer'))
    return
  }
  setTimeout(() => {
    const id = Math.random().toString(36).split('.')[1].slice(0, 4)
    cb(null, { id })
  }, 300)
}
```

This API mimics some kind of async storage mechanism, such as to a database. In some circumstances it is infeasible to check for a specific value (for instance an ID returned from a database). For those cases, we can check for the presence of an ID, or apply some validation. In our case we can at least check that the length of the ID is 4.

Write tests for the `store.js` file. Ensure that when `npm test` is executed with the labs-2 folder as the current working directory the `store.js` file is fully tested.

Any additional dependencies, such as a test harness, may be additionally installed.

Also in the labs-2 folder is a `validate.js` file. The implementation can be checked with `node validate.js`. The implementation is successful if the final output of `node validate.js` is `passed!`.

For completeness the following is the `validate.js` file contains the following, but it is not necessary to understand it for the purposes of this exercise:

```
'use strict'
const assert = require('assert').strict
const { spawnSync } = require('child_process')
const { writeFileSync } = require('fs')
const store = require.resolve('./store')
const storeCode =
Buffer.from('2775736520737472696374270a6d6f64756c652e6578706f727473203
d202876616c75652c20636229203d3e207b0a2020696620284275666665722e6973427
5666665722876616c756529203d3d3d2066616c736529207b0a2020202063622845727
26f722827696e707574206d75737420626520612062756666657229290a202020207
2657475726e0a20207d0a2020736574546696d656f757428282929203d3e207b0a2020202
0636f6e7374206964203d204d6174682e72616e646f6d28292e746f537472696e67283
336292e73706c697428272e27295b315d2e736c69636528302c2034290a2020202020636
2286e756c6c2c207b206964207d290a20207d2c20333030290a7d0a', 'hex')

try {
  {
    writeFileSync(store, storeCode)

    const sp = spawnSync('npm', ['test'], {
      stdio: 'ignore'
    })
    assert.equal(sp.status, 0, 'tests should be successful (is
package.json test script configured?)')
  }

  {
    const badOutput = `'use strict'
      module.exports = (value, cb) => {
        if (Buffer.isBuffer(value) === false) {
          cb(Error('input must be a buffer'))
          return
        }
        setTimeout(() => {
          const id = Math.random().toString(36).split('.')[1].slice(0,
2)
          cb(null, { id })
        }, 300)
      }

    `

    writeFileSync(store, badOutput)

    const sp = spawnSync('npm', ['test'], {
      stdio: 'ignore'
    })

    assert.equal(sp.status, 1, 'output should be tested (id length)')
  }

  {
    const badValidation = `'use strict'
    module.exports = (value, cb) => {
      if (Buffer.isBuffer(value) === true) {
        cb(Error('input must be a buffer'))
        return
      }
      setTimeout(() => {
        const id = Math.random().toString(36).split('.')[1].slice(0,
4)
        cb(null, { id })
      }, 300)
    }

    `

    writeFileSync(store, badValidation)

    const sp = spawnSync('npm', ['test'], {
      stdio: 'ignore'
    })

    assert.equal(sp.status, 1, 'error case should be tested')
  }

  {
    const unexpectedError = `'use strict'
    module.exports = (value, cb) => {
      cb(Error('input must be a buffer'), {id: '1234'})
    }
    `

    writeFileSync(store, unexpectedError)

    const sp = spawnSync('npm', ['test'], {
      stdio: 'ignore'
    })

    assert.equal(sp.status, 1, 'unexpected errors should be guarded -
e.g. ifError')
  }

  console.log('passed!')
} finally {
  writeFileSync(store, storeCode)
}
```

THE LINUX FOUNDATION | Training & Certification

Lab 16.3 - Test a Promise-Based async/await API

The labs-3 folder contains a `package.json` file and a `store.js` file.

The `package.json` file contains the following:

```
{
  "name": "labs-3",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "UNLICENSED"
}
```

The `store.js` file contains the following:

```
'use strict'
const { promisify } = require('util')
const timeout = promisify(setTimeout)
module.exports = async (value) => {
  if (Buffer.isBuffer(value) === false) {
    throw Error('input must be a buffer')
  }
  await timeout(300)
  const id = Math.random().toString(36).split('.')[1].slice(0, 4)
  return { id }
}
```

This API mimics some kind of async storage mechanism, such as to a database. In some circumstances it is infeasible to check for a specific value (for instance an ID returned from a database). For those cases, we can check for the presence of an ID, or apply some validation. In our case we can at least check that the length of the ID is 4.

Write tests for the `store.js` file. Ensure that when `npm test` is executed with the labs-2 folder as the current working directory the `store.js` file is fully tested.

Any additional dependencies, such as a test harness, may be additionally installed.

Also in the labs-3 folder is a `validate.js` file. The implementation can be checked with `node validate.js`. The implementation is successful if the final output of `node validate.js` is `passed!`.

For completeness the following is the `validate.js` file contains the following, but it is not necessary to understand it for the purposes of this exercise:

```
'use strict'
const assert = require('assert').strict
const { spawnSync } = require('child_process')
const { writeFileSync } = require('fs')
const store = require.resolve('./store')
const storeCode =
Buffer.from('2775736520737472696374270a636f6e7374207b2070726f6d6973696
679207d203d207265717569726528277574696c27290a636f6e73742074696d656f757
4203d2070726f6d69736966792873657454696d656f7574290a6d6f64756c652e65787
06f727473203d206173796e63202876616c756529203d3e207b0a20206966202842756
66665722e697342756666657228766616c756529203d3d2066616c736529207b0a202
020207468726f77204572726f722827696e707574206d7573742062652061206275666
6657227290a20207d0a2020617761697420074696d656f75742833303290a2020636f6
e7374206964203d204d6174682e72616e646f6d28292e746f537472696e67283336292
e73706c697428272e27295b315d2e736c69636528302c2034290a20207265747572 6e2
07b206964207d0a7d0a', 'hex')

try {
  {
    writeFileSync(store, storeCode)
    const sp = spawnSync('npm', ['test'], {
      env: { ...process.env, NODE_OPTIONS:
'--unhandled-rejections=strict' },
      stdio: 'ignore'
    })

    assert.equal(sp.status, 0, 'tests should be successful (is
package.json test script configured?)')
  }

  {
    const badOutput = `'use strict'
      const { promisify } = require('util')
      const timeout = promisify(setTimeout)
      module.exports = async (value) => {
        if (Buffer.isBuffer(value) === false) {
          throw Error('input must be a buffer')
        }
        await timeout(300)
        const id = Math.random().toString(36).split('.')[1].slice(0,
2)
        return { id }
      }
    `

    writeFileSync(store, badOutput)

    const sp = spawnSync('npm', ['test'], {
      env: { ...process.env, NODE_OPTIONS:
'--unhandled-rejections=strict' },
      stdio: 'ignore'
    })
    assert.equal(sp.status, 1, 'output should be tested (id length)')
  }

  {
    const badValidation = `'use strict'
      const { promisify } = require('util')
      const timeout = promisify(setTimeout)
      module.exports = async (value) => {
        await timeout(300)
        const id = Math.random().toString(36).split('.')[1].slice(0,
4)
        return { id }
      }
    `

    writeFileSync(store, badValidation)

    const sp = spawnSync('npm', ['test'], {
      env: { ...process.env, NODE_OPTIONS:
'--unhandled-rejections=strict' },
      stdio: 'ignore'
    })

    assert.equal(sp.status, 1, 'error case should be tested')
  }

  console.log('passed!')
} finally {
  writeFileSync(store, storeCode)
}
```