Node.js Services Development (LFW212)

Lab Exercises (June 2023)

Table of Contents

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:

As long as the first number (followed by "v") is 20, 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:

If the first number before the period is 10 or greater, then we are ready to proceed.

Lab 3.1 - Deliver Data from a Library API

The labs-1 folder contains the following files:

- **data.js**
- **package.json**
- **validate.js**

The **data.js** file contains the following:

```
'use strict'
const { promisify } = require('util')
const { randomBytes } = require('crypto')
const timeout = promisify(setTimeout)

async function data () {
  await timeout(50)
  return randomBytes(10).toString('base64')
}

module.exports = data
```

The **data.js** file exports a function that returns a promise (an **async** function) that resolves to a random BASE64 string. This function represents some kind of asynchronous data source.

The **package.json** file contains the following:

```
{
  "name": "labs-1",
  "scripts": {
```

```
    "start": "echo \"TODO: SET THE START SCRIPT\" && exit 1"
  }
}
```

Using any Node core library and/or web framework create an HTTP server that meets the following criteria:

- Listens on localhost
- Listens on port 3000
- Responds to HTTP GET requests to **/** with data from the **data** function as exported from the **data.js**
- Responds with a 404 to GET requests to any other route

The **package.json start** script must contain a command to start the server.

Run the following command to check whether the created server meets the criteria:

node validate

If the server was correctly implemented, the output of this command should be as follows:

Lab 3.2 - Implement a Status Code Response

The labs-2 folder contains the following files:

- **package.json**
- **validate.js**

The **package.json** file contains the following:

```
{
  "name": "labs-2",
  "scripts": {
    "start": "echo \"TODO: SET THE START SCRIPT\" && exit 1"
  }
}
```

Using any Node core library and/or web framework create an HTTP server that meets the following criteria:
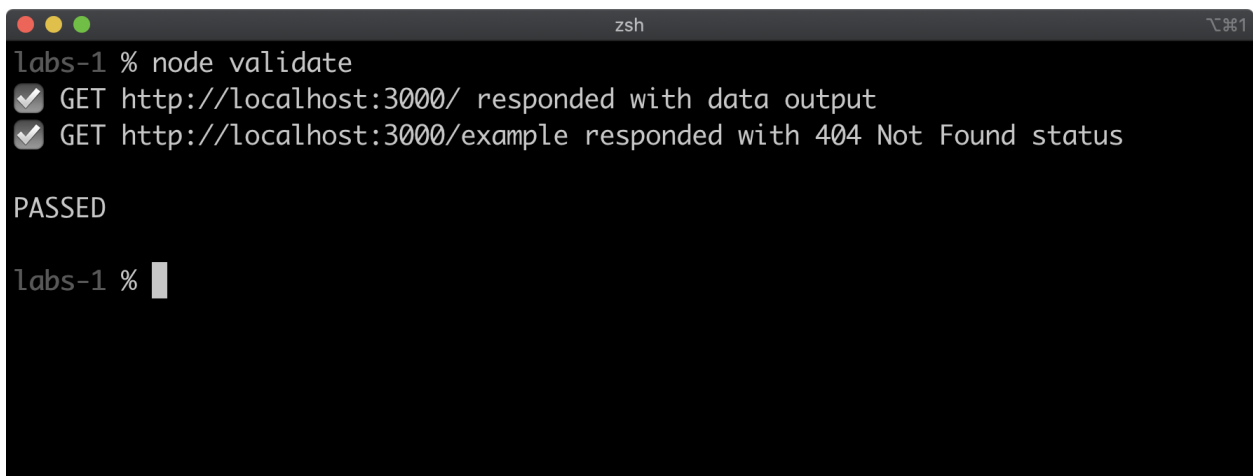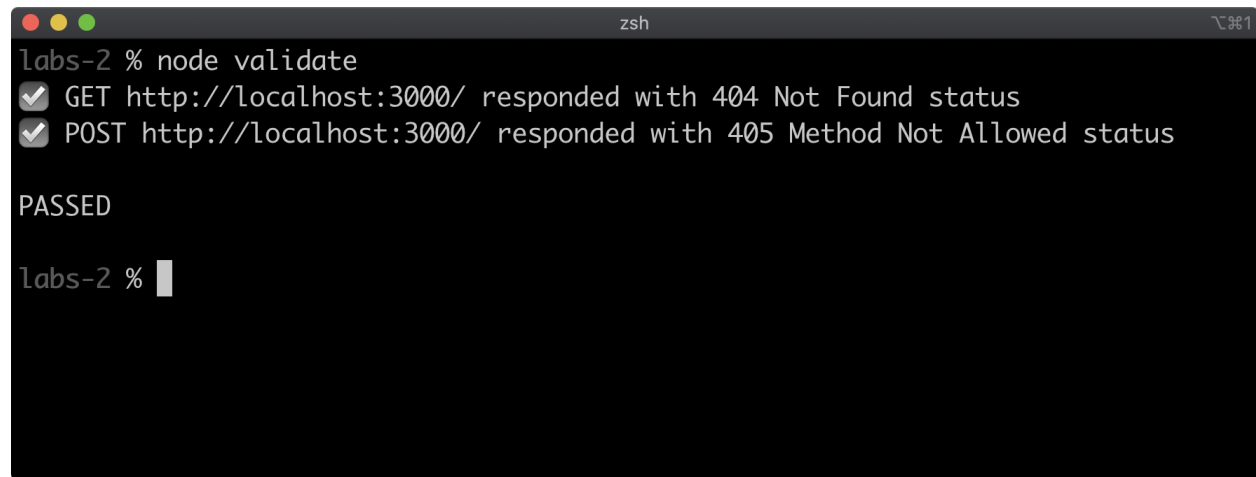
- Listens on localhost
- Listens on port 3000
- Responds to HTTP GET requests to **/** with a 200 OK HTTP status, the content is irrelevant
- Responds to HTTP POST requests to **/** with a 405 Method Not Allowed HTTP status

The **package.json start** script must contain a command to start the server.

Run the following command to check whether the created server meets the criteria:

```
node validate
```

If the server was correctly implemented, the output of this command should be as follows:

```
labs-2 % node validate
✅ GET http://localhost:3000/ responded with 404 Not Found status
✅ POST http://localhost:3000/ responded with 405 Method Not Allowed status

PASSED

labs-2 %
```

THE **LINUX** FOUNDATION | Training & Certification

Lab 4.1 - Render a View

Using either Fastify or Express, create a project with a **/me** route. Render a view that uses the **layout.hbs** that was created in this chapter to create a small profile page. The HTML content is unimportant, just make sure to render a view.

The labs-1 folder contains a file named **validate.js**. Make sure the server that the **/me** route was added to is running in another terminal, then with the current working directory set to the **labs-1** folder, run the following to check the implementation:

node validate

If successful the following output should be seen:

```
labs-1 % node validate.js
✅ GET http://localhost:3000/me responded with 200 response
✅ GET http://localhost:3000/me reuses the layout.hbs view to render

PASSED

labs-1 %
```

Lab 4.2 - Stream Some Content

The following code creates a stream with a built in delay when the **stream** function is called:

```
const { Readable, Transform } = require('stream')

function stream () {
  const readable = Readable.from([
    'this', 'is', 'a', 'stream', 'of', 'data'
  ].map((s) => s + '<br>'))
  const delay = new Transform(({
    transform (chunk, enc, cb) {
      setTimeout(cb, 500, null, chunk)
    }
  }))
  return readable.pipe(delay)
}
```

Using either Fastify or Express, create a new route at path **/data** and send the data from this stream to the response when the **/data** route is requested.

The labs-2 folder contains a file named **validate.js**. Make sure the server that the **/data** route was added to is running in another terminal, then with the current working directory set to the **labs-2** folder, run the following to check the implementation:

```
node validate
```

If successful the following output should be the result of this command:

```
                                          zsh                                    ⌥⌘1
labs-2 % node validate.js
✅ GET http://localhost:3000/data responded with 200 response
✅ GET http://localhost:3000/data has expected delay between items in repsonse stream

PASSED

labs-2 %
```

Lab 5.1 - Implement a RESTful JSON GET

The labs-1 folder contains the following files:

- **model.js**
- **package.json**
- **validate.js**

The **start** field of the **package.json** file looks as follows:

```
"start": "echo \"Error: start script not specified\" && exit 1",
```

The **model.js** file contains the following:

```
'use strict'

module.exports = {
  boat: boatModel()
}

function boatModel () {
  const db = {
    1: { brand: 'Chaparral', color: 'red' },
    2: { brand: 'Chaparral', color: 'blue' }
  }

  return {
    uid,
    create,
    read,
    update,
```

```
    delete: del
}

function uid () {
  return Object.keys(db)
    .sort((a, b) => a - b)
    .map(Number)
    .filter((n) => !isNaN(n))
    .pop() + 1 + ''
}

function create (id, data, cb) {
  if (db.hasOwnProperty(id)) {
    const err = Error('resource exists')
    err.code = 'E_RESOURCE_EXISTS'
    setImmediate(() => cb(err))
    return
  }
  db[id] = data
  setImmediate(() => cb(null, id))
}

function read (id, cb) {
  if (!(db.hasOwnProperty(id))) {
    const err = Error('not found')
    err.code = 'E_NOT_FOUND'
    setImmediate(() => cb(err))
    return
  }
  setImmediate(() => cb(null, db[id]))
}

function update (id, data, cb) {
  if (!(db.hasOwnProperty(id))) {
    const err = Error('not found')
    err.code = 'E_NOT_FOUND'
    setImmediate(() => cb(err))
    return
  }
  db[id] = data
  setImmediate(() => cb())
}
```

```
function del (id, cb) {
  if (!(db.hasOwnProperty(id))) {
    const err = Error('not found')
    err.code = 'E_NOT_FOUND'
    setImmediate(() => cb(err))
    return
  }
  delete db[id]
  setImmediate(() => cb())
}
}
```

Use either Fastify or Express to implement a RESTful HTTP server so that when the command **npm start** is executed, a server is started that listens on **process.env.PORT**.

If implementing in Fastify, remember that running **npm init fastify -- --integrate** in the labs-1 folder will set up the project **npm start** is executed the server will automatically listen on **process.env.PORT**.

The server should support a **GET** request to a single route: **/boat/{id}** where **{id}** is a placeholder for any given ID - for instance **/boat/2**.

The **GET /boat/{id}** route should respond with a JSON payload. The route should also respond with the correct headers for a JSON response (**Content-Type: application/json**).

The server should only support this GET route. That means that any other routes or any other verbs should be handled according to the HTTP specification. Thankfully Express and Fastify will do most of this for us.

The following cases must be successfully handled:

- A successful request should respond with a **200** status code. Express and Fastify do this automatically.
- The response should have the correct mime type header. In this case we need to make sure the **Content-Type** header is set to **application/json**.
- A GET request to a route that does not exist should respond with a **404** status code. Fastify does this automatically and the typical Express configuration also handles this by default.
- If a given boat ID isn't found in the model the server should respond with a **404** status code. The response body can contain anything, but it's important that the response status is set to 404.
- Unexpected errors in the model should cause the server to respond with a **500** status code. This means that if the **read** method of the model passed an Error object to the

callback that was unexpected or unrecognized, that error needs to be propagated to the framework we're using in some way so that the framework can automatically generate a 500 response.
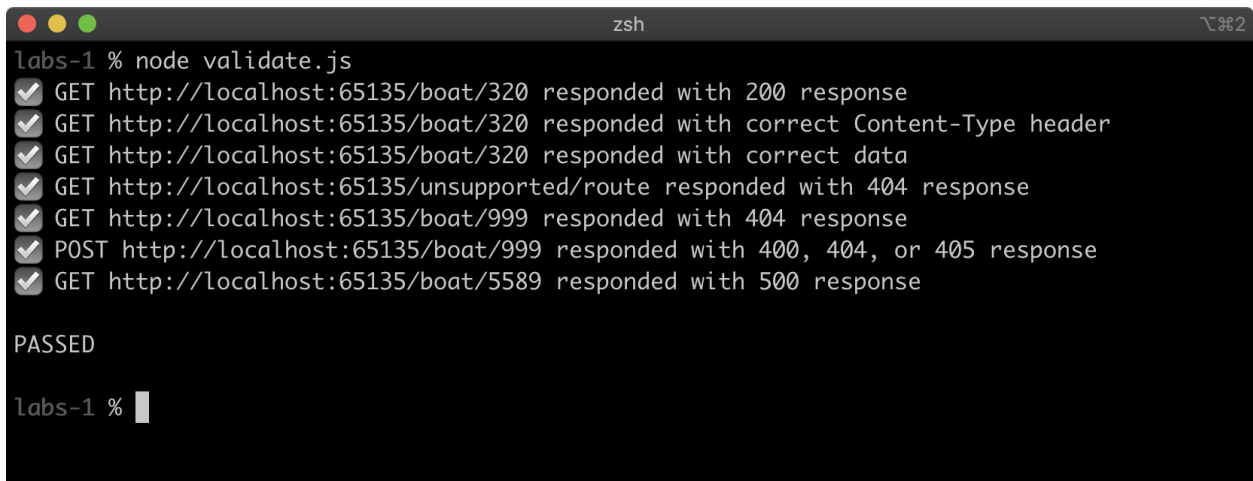
- In the HTTP specification there is some ambiguity over how to handle unsupported HTTP methods. Any HTTP method other than **GET** should be responded to with either a **400**, **404** or **405** status code. Again Fastify and Express will respond to unsupported methods with one of these status codes.

Do not edit the **model.js** file, it will be overwritten by the validation process anyway. The **model.js** file is deliberately noisy, providing methods that we don't need for this exercise. This reflects the philosophical approach of the certification to provide occasionally messy API's to integrate with in order to better reflect real-world scenarios.

Once the server has been implemented, the following command can be executed to validate the implementation:

node validate.js

When correctly implemented the result of this command should be as follows:

```
labs-1 % node validate.js
✅ GET http://localhost:65135/boat/320 responded with 200 response
✅ GET http://localhost:65135/boat/320 responded with correct Content-Type header
✅ GET http://localhost:65135/boat/320 responded with correct data
✅ GET http://localhost:65135/unsupported/route responded with 404 response
✅ GET http://localhost:65135/boat/999 responded with 404 response
✅ POST http://localhost:65135/boat/999 responded with 400, 404, or 405 response
✅ GET http://localhost:65135/boat/5589 responded with 500 response

PASSED

labs-1 %
```

Lab 6.1 - Implement a RESTful JSON POST

The labs-1 folder contains the following files:

- **model.js**
- **package.json**
- **validate.js**

The **model.js** file and the **package.json** file are exactly the same as the first lab exercise in the previous chapter.

The **model.js** file has the following content:

```
'use strict'

module.exports = {
  boat: boatModel()
}

function boatModel () {
  const db = {
    1: { brand: 'Chaparral', color: 'red' },
    2: { brand: 'Chaparral', color: 'blue' }
  }

  return {
    uid,
    create,
    read,
    update,
    del
```

```
}

function uid () {
  return Object.keys(db)
    .sort((a, b) => a - b)
    .map(Number)
    .filter((n) => !isNaN(n))
    .pop() + 1 + ''
}

function create (id, data, cb) {
  if (db.hasOwnProperty(id)) {
    const err = Error('resource exists')
    err.code = 'E_RESOURCE_EXISTS'
    setImmediate(() => cb(err))
    return
  }
  db[id] = data
  setImmediate(() => cb(null, id))
}

function read (id, cb) {
  if (!(db.hasOwnProperty(id))) {
    const err = Error('not found')
    err.code = 'E_NOT_FOUND'
    setImmediate(() => cb(err))
    return
  }
  setImmediate(() => cb(null, db[id]))
}

function update (id, data, cb) {
  if (!(db.hasOwnProperty(id))) {
    const err = Error('not found')
    err.code = 'E_NOT_FOUND'
    setImmediate(() => cb(err))
    return
  }
  db[id] = data
  setImmediate(() => cb())
}
```

```
function del (id, cb) {
  if (!(db.hasOwnProperty(id))) {
    const err = Error('not found')
    err.code = 'E_NOT_FOUND'
    setImmediate(() => cb(err))
    return
  }
  delete db[id]
  setImmediate(() => cb())
  }
}
```

The `package.json` file looks as follows:

```
{
  "name": "labs-1",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "start": "echo \"Error: start script not specified\" && exit 1",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Use either Fastify or Express to implement a RESTful HTTP server so that when the command `npm start` is executed it starts a server that listens on `process.env.PORT`.

The server should support a `POST` request to `/boat` that uses the `model.js` file to create a new entry. The route should only accept `application/json` mime-type requests and should respond with `application/json` content-type responses.

The `POST` request should expect JSON data to be sent in the following format:

```
{ data: { brand, color } }
```

A successful request should respond with a 201 Created status code. Unexpected errors should result in a 500 Server Error response.

The service must also support the same `GET /boat/{id}` route as implemented in the previous chapter.
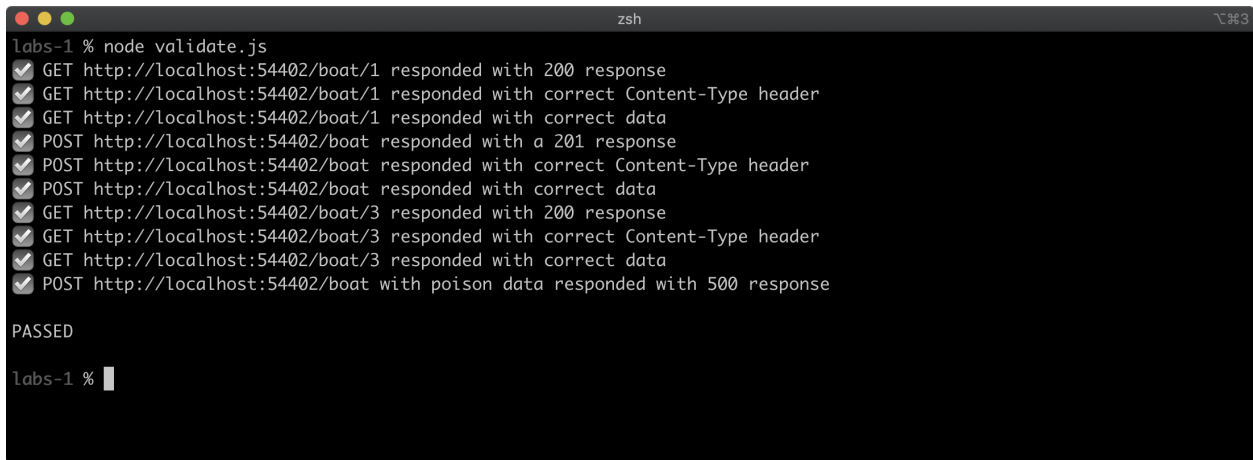
It is not necessary to validate user input for this exercise.

Feel free to copy the files and folders from the labs-1 answer of the previous chapter into the labs-1 folder of this chapter and then build upon, or else start from scratch, as preferred.

Making sure that the labs-1 folder is the current working directory, run the following command to validate the completed exercise:

`node validate`

When correctly implemented, this command should output the following:

```
labs-1 % node validate.js
✅ GET http://localhost:54402/boat/1 responded with 200 response
✅ GET http://localhost:54402/boat/1 responded with correct Content-Type header
✅ GET http://localhost:54402/boat/1 responded with correct data
✅ POST http://localhost:54402/boat responded with a 201 response
✅ POST http://localhost:54402/boat responded with correct Content-Type header
✅ POST http://localhost:54402/boat responded with correct data
✅ GET http://localhost:54402/boat/3 responded with 200 response
✅ GET http://localhost:54402/boat/3 responded with correct Content-Type header
✅ GET http://localhost:54402/boat/3 responded with correct data
✅ POST http://localhost:54402/boat with poison data responded with 500 response

PASSED

labs-1 %
```

Lab 6.2 - Implement a RESTful JSON DELETE

The labs-2 folder contains the following files:

- **model.js**
- **package.json**
- **validate.js**

The **model.js** file and the **package.json** file are exactly the same as the first lab exercise in this chapter and in the previous chapter.

The **model.js** file has the following content:

```
'use strict'

module.exports = {
  boat: boatModel()
}

function boatModel () {
  const db = {
    1: { brand: 'Chaparral', color: 'red' },
    2: { brand: 'Chaparral', color: 'blue' }
  }

  return {
    uid,
    create,
    read,
    update,
    del
  }
```

```
function uid () {
  return Object.keys(db)
    .sort((a, b) => a - b)
    .map(Number)
    .filter((n) => !isNaN(n))
    .pop() + 1 + ''
}

function create (id, data, cb) {
  if (db.hasOwnProperty(id)) {
    const err = Error('resource exists')
    err.code = 'E_RESOURCE_EXISTS'
    setImmediate(() => cb(err))
    return
  }
  db[id] = data
  setImmediate(() => cb(null, id))
}

function read (id, cb) {
  if (!(db.hasOwnProperty(id))) {
    const err = Error('not found')
    err.code = 'E_NOT_FOUND'
    setImmediate(() => cb(err))
    return
  }
  setImmediate(() => cb(null, db[id]))
}

function update (id, data, cb) {
  if (!(db.hasOwnProperty(id))) {
    const err = Error('not found')
    err.code = 'E_NOT_FOUND'
    setImmediate(() => cb(err))
    return
  }
  db[id] = data
  setImmediate(() => cb())
}
function del (id, cb) {
  if (!(db.hasOwnProperty(id))) {
    const err = Error('not found')
```

```
        err.code = 'E_NOT_FOUND'
        setImmediate(() => cb(err))
        return
      }
      delete db[id]
      setImmediate(() => cb())
  }
}
```

The `package.json` file looks as follows:

```
{
  "name": "labs-1",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "start": "echo \"Error: start script not specified\" && exit 1",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Use either Fastify or Express to implement a RESTful HTTP server so that when the command `npm start` is executed it starts a server that listens on `process.env.PORT`.

The server should support a *DELETE* request to `/boat/{id}` where `{id}` is a placeholder for any given ID - for instance `/boat/2`.

A successful request should result in an empty response body with a 204 No Content status code. If the specified ID does not exist, the response should have a 404 status code. Any unexpected errors should result in a 500 Server Error response.

The service must also support the same `GET /boat/{id}` route as implemented in the previous chapter.

Feel free to copy the files and folders from the labs-1 answer into this labs-2 answer or start from scratch as preferred.

Making sure that the labs-2 folder is the current working directory, run the following command to validate the completed exercise:

`node validate`

Lab 7.1 - Implement a Data Aggregating Service

The labs-1 folder contains the following files:

- **package.json**
- **boat-service.js**
- **brand-service.js**
- **validate.js**

The **package.json** file has the following content:

```
{
  "name": "labs-1",
  "version": "1.0.0",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "echo \"Error: start script not specified\" && exit 1"
  },
  "keywords": [],
  "license": "UNLICENSED"
}
```

Create a service that is started when the **npm start** command is executed that consumes two other HTTP services.

The service must bind to a port number defined by the **PORT** environment variable.

The services are provided with mock data in this project folder as **boat-service.js** and **brand-service.js**.

When started, each mock service outputs a port. This output can be used to set the
BOAT_SERVICE_PORT and **BRAND_SERVICE_PORT** environment variables when starting the
aggregating service.

For instance if the port of the Boat service is 3333 and the port of the Brand service is 3334 the
server can be started like so:

```
PORT=3000 BOAT_SERVICE_PORT=3333 BRAND_SERVICE_PORT=3334 npm start
```

Be sure to use the **BOAT_SERVICE_PORT** and **BRAND_SERVICE_PORT** environment variables
in the service to get the relevant port for each service. For example, the values of these
environment variables could be loaded into the service implementation like so:

```
const {
  BOAT_SERVICE_PORT,
  BRAND_SERVICE_PORT
} = process.env
```

To make a request to the Brand service:
`http://localhost:[BRAND_SERVICE_PORT]/[id]`

The Boat service responds with JSON data in the following format:

```
{
  "id": Number,
  "brand": Number,
  "color": String
}
```

A request to the Boat service: `http://localhost:[BOAT_SERVICE_PORT]/[id]`
The `id` and `brand` properties will only ever be Integers.

The Brand service responds with JSON data in the following format:

```
{
  "id": Number,
  "name": String
}
```

The **brand** property of the Boat service output corresponds to the **id** of the Brand service entities.

Create a service which accepts GET requests at **http://localhost:[PORT]/[id]**. Use the incoming **id** from the GET request to make a request to the Boat service and use data from the Boat service to make a request to the Brand service to retrieve associated brand data.

Combine the information from the two responses into a JSON payload and send that as a response. The JSON payload should have the following form:

```
{
  "id": Number,
  "color": String,
  "brand": String
}
```

The aggregating service should handle various scenarios in the following ways:

- For a normal successful request, respond with a 200 status code and the JSON payload of combined data (**{id, color, brand}**) as the response body. The **Content-Type** header must be **application/json**.
- Respond with a 404 status code if either service responds with a 404 status.
- If either service is not available, respond with a 500 status code with any response body.
- If either service responds with a non-200 status code then respond with a 500 status code with any response body.
- If a request is made to the aggregating service with an ID that is not a valid integer, respond with a 400 status code, the response body is unimportant and can be anything.
- If either service responds with a 4XX status code that is not a 400 or 404 (401-403, 405-499) status code, then respond with a 500 status code with any response body.
- Be sure that if an upstream service is not available, that the service responds within 1250ms.

The validator code for this exercise starts the services automatically. Make sure that the services are not running and then run the following command in the labs-1 folder to check the implementation:

node validate.js

If the aggregating service is successfully implemented this should result in the following output:

```
● ● ●                                    zsh                                    ⌥⌘3
labs-1 % node validate.js

> labs-1@1.0.0 start /Users/davidclements/JSNSD-course/labs/ch-7/labs-1
> node app

✅ GET http://localhost:3000/1 responded with 200 response
✅ GET http://localhost:3000/1 responded with correct Content-Type header
✅ GET http://localhost:3000/1 responded with correct data
✅ GET http://localhost:3000/2 responded with 404 response
✅ GET http://localhost:3000/3 responded with 404 response
✅ GET http://localhost:3000/boat responded with 400
✅ GET http://localhost:3000/1 responded with 500 response (brand service is down)
✅ GET http://localhost:3000/1 responded with 200 response
✅ GET http://localhost:3000/1 responded with correct Content-Type header
✅ GET http://localhost:3000/1 responded with correct data
✅ GET http://localhost:3000/1 responded with 500 response (boat service is down)
✅ GET http://localhost:3000/1 responded with 200 response
✅ GET http://localhost:3000/1 responded with correct Content-Type header
✅ GET http://localhost:3000/1 responded with correct data
✅ GET http://localhost:3000/1 responded with 500 response (both services are down)
✅ GET http://localhost:3000/1 responded with 200 response
✅ GET http://localhost:3000/1 responded with correct Content-Type header
✅ GET http://localhost:3000/1 responded with correct data

PASSED

labs-1 % █
```

Lab 8.1 - Implement an HTTP Route-Based Proxy

The labs-1 folder contains the following files:

- **package.json**
- **validate.js**

The **package.json** file has the following content:

```
{
  "name": "labs-1",
  "version": "1.0.0",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "echo \"Error: start script not specified\" && exit 1"
  },
  "keywords": [],
  "license": "UNLICENSED"
}
```

Create an HTTP service that initializes when `npm start` is executed and listens on whatever the `PORT` environment variable is set to.

The service must be a transparent reverse HTTP proxy server such that a request to `http://localhost:{PORT}/?url={URL}` will respond with:

1. the status code of `{URL}`
2. the headers provided at `{URL}`
3. the contents of the body at `{URL}`

The **{URL}** will only ever hold HTTP URLs, there's no need to proxy HTTPS URLs.
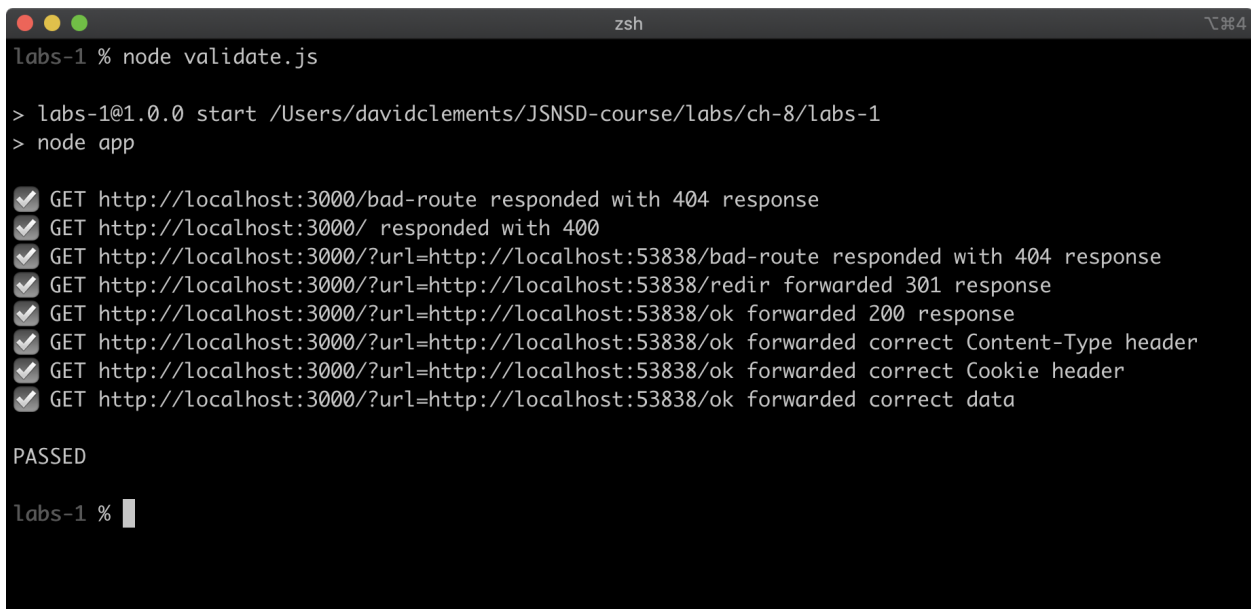
The service must meet the following conditions:

- A request to any route other than **/** should respond with an HTTP **Not Found** response.
- A request to **/** without a **url** query-string parameter should result in a **Bad Request** HTTP response.
- The proxy only needs to support HTTP GET requests.

Run the following command to check whether the implementation was successful:

node validate.js

When successfully implemented the output should be similar to the following:

```
labs-1 % node validate.js

> labs-1@1.0.0 start /Users/davidclements/JSNSD-course/labs/ch-8/labs-1
> node app

✅ GET http://localhost:3000/bad-route responded with 404 response
✅ GET http://localhost:3000/ responded with 400
✅ GET http://localhost:3000/?url=http://localhost:53838/bad-route responded with 404 response
✅ GET http://localhost:3000/?url=http://localhost:53838/redir forwarded 301 response
✅ GET http://localhost:3000/?url=http://localhost:53838/ok forwarded 200 response
✅ GET http://localhost:3000/?url=http://localhost:53838/ok forwarded correct Content-Type header
✅ GET http://localhost:3000/?url=http://localhost:53838/ok forwarded correct Cookie header
✅ GET http://localhost:3000/?url=http://localhost:53838/ok forwarded correct data

PASSED

labs-1 %
```

Lab 8.2 - Implement a Full Proxying Service

The labs-2 folder is completely empty. Create a service that listens on port 3000.
The service must proxy all requests/responses to https://jsonplaceholder.typicode.com.

Use the following command to check whether the implementation was successful:

```
node -e "http.get('http://localhost:3000/todos/1', (res) =>
res.pipe(process.stdout))"
```

This should output something similar to the following:

```
labs-2 % node -e "http.get('http://localhost:3000/todos/1', (res) => res.pipe(process.stdout))"
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
labs-2 %
```

Lab 9.1 - Implement a Service That Is Not Vulnerable to Parameter Pollution

The labs-1 folder contains the following files:

- **package.json**
- **app.js**
- **validate.js**

The **package.json** file contains the following:

```
{
  "name": "labs-1",
  "version": "1.0.0",
  "scripts": {
    "start": "node app.js"
  },
  "license": "UNLICENSED",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

Note that Express is a dependency of the project. Install the project dependency with the following command, executed within the labs-1 folder:

```
npm install
```

The **app.js** file contains the following:

```
'use strict'
const express = require('express')
const app = express()
const router = express.Router()
const { PORT = 3000 } = process.env

router.get('/', (req, res) => {
  setTimeout(() => {
    res.send((req.query.un || '').toUpperCase())
  }, 1000)
})

app.use(router)

app.listen(PORT, () => {
  console.log(`Express server listening on ${PORT}`)
})
```

This is a small Express service that uppercases any input sent via a **un** query string parameter, but it waits one second before sending the response.

This service is vulnerable to parameter pollution. A URL such as http://localhost:3000/?un=a&un=b will cause the service to crash, assuming the service is listening on port 3000.

Fix it, without changing any of the current functionality.

The parameter pollution attack may be handled as seen fit. For instance upper casing all forms, or sending a 400 Bad Request, or any kind of response. The only thing that must not happen is the service crashing and requests containing query-strings with a single **un** parameter must continue to respond with the uppercased version of that value.

Run the **validate.js** file as follows, to validate the fix:

node validate.js

If successful this should output something similar to the following:

```
● ● ●                                         zsh                                        ⌥⌘5
labs-1 % node validate.js

> labs-1@1.0.0 start /Users/davidclements/JSNSD-course/labs/ch-9/labs-1
> node app.js

Express server listening on 3000
✅ GET http://localhost:3000/?un=xx8f0e22 responded with 200 response
✅ GET http://localhost:3000/?un=xx8f0e22 responded after approx. 1s
✅ GET http://localhost:3000/?un=xx8f0e22 responded with correct data
✅ GET http://localhost:3000/?un=xxedd536&un=xx7139b3 responded without service crashing
✅ GET http://localhost:3000 responded without service crashing

PASSED

labs-1 % ▊
```

Lab 9.2 - Validate a POST Request

The labs-2 folder contains a Fastify service. Other than the typical Fastify directory structure, it has a **model.js** file, a **routes/boat/index.js**, a **package-lock.json** file and a **validate.js** file.

The **model.js** file is the same as from the labs of Chapters 5 and 6.

The **routes/boat/index.js** file supports the following routes:

- **POST /boat**
- **GET /boat/{id}**
- **DELETE /boat/{id}**

Apply validation to the POST route request body so that any POST request bodies that do not have the shape **{ data: { brand, color }}** are rejected with a 400 Bad Request status code. Additional properties are allowed, but should be stripped before being stored.

Do not remove or otherwise modify any of the routes.

Remember to run **npm install** in the labs-2 folder to install project dependencies.

Run the **validate.js** file as follows, to validate that the route validation was correctly implemented:

node validate.js

If successful this should output something similar to the following:

```
●  ●  ●                                    zsh                                        ⌥⌘3
labs-2 % node validate.js
☑ GET http://localhost:63326/boat/1 responded with 200 response
☑ GET http://localhost:63326/boat/1 responded with correct Content-Type header
☑ GET http://localhost:63326/boat/1 responded with correct data
☑ POST http://localhost:63326/boat responded with a 201 response
☑ POST http://localhost:63326/boat responded with correct Content-Type header
☑ POST http://localhost:63326/boat responded with correct data
☑ GET http://localhost:63326/boat/3 responded with 200 response
☑ GET http://localhost:63326/boat/3 responded with correct Content-Type header
☑ GET http://localhost:63326/boat/3 responded with correct data
☑ POST http://localhost:63326/boat responded with a 400 response
☑ POST http://localhost:63326/boat responded with a 400 response
☑ POST http://localhost:63326/boat with poison data responded with 500 response

PASSED

labs-2 % ▊
```

Lab 10.1 - Block an Attackers IP Address with Express

The labs-1 folder contains an Express application along with a **validate.js** file.

Imagine this is a deployed service, which is receiving a DoS attack from the IP address 111.34.55.211.

Edit the service so that this IP address, and only this IP address, receives a 403 Forbidden response from the service.

Execute the following command to check the mitigation step worked:

node validate.js

If successful, output similar to the following should be seen:

```
labs-1 % node validate.js

> labs-1@0.0.0 start /Users/davidclements/JSNSD-course/labs/ch-10/labs-1
> node ./bin/www

GET / 200 120.821 ms - 170
✅ GET http://localhost:3000/ responded with 200 response
✅ GET http://localhost:3000/ responded with 403 when requested from attacker IP

PASSED

labs-1 %
```

Lab 10.2 - Block an Attackers IP Address with Fastify

The labs-2 folder contains a Fastify application along with a `validate.js` file.

Imagine this is a deployed service, which is receiving a DoS attack from the IP address 211.133.33.113.

Edit the service so that this IP address, and only this IP address, receives a 403 Forbidden response from the service.

Execute the following command to check the mitigation step worked:

`node validate.js`

If successful, output similar to the following should be seen:

```
● ● ●                              zsh                              ⌥⌘5
labs-2 % node validate.js

> labs-2@1.0.0 dev /Users/davidclements/JSNSD-course/labs/ch-10/labs-2
> fastify start -w -l info -P app.js

21:55:56 ✨ Server listening at http://127.0.0.1:3000
21:55:56 ✨ incoming request GET xxx /
21:55:56 ✨ request completed 2ms
☑ GET http://localhost:3000/ responded with 200 response
21:55:56 ✨ incoming request GET xxx /
21:55:56 ✨ forbidden
21:55:56 ✨ request completed 1ms
☑ GET http://localhost:3000/ responded with 403 when requested from attacker IP

PASSED

labs-2 % ▮
```