

a Structure of backend application, introduction to testing

Let's continue our work on the backend of the notes application we started in [part 3](#).

Project structure

Before we move into the topic of testing, we will modify the structure of our project to adhere to Node.js best practices.

After making the changes to the directory structure of our project, we end up with the following structure:

```
├─ index.js
├─ app.js
├─ build
│   └─ ...
├─ controllers
│   └─ notes.js
├─ models
│   └─ note.js
├─ package-lock.json
├─ package.json
├─ utils
│   ├── config.js
│   ├── logger.js
│   └─ middleware.js
```

So far we have been using `console.log` and `console.error` to print different information from the code. However, this is not a very good way to do things. Let's separate all printing to the console to

its own module *utils/logger.js*:

```
const info = (...params) => {
  console.log(...params)
}

const error = (...params) => {
  console.error(...params)
}

module.exports = {
  info, error
}
```

The logger has two functions, *info* for printing normal log messages, and *error* for all error messages.

Extracting logging into its own module is a good idea in more ways than one. If we wanted to start writing logs to a file or send them to an external logging service like graylog or papertrail we would only have to make changes in one place.

The contents of the *index.js* file used for starting the application gets simplified as follows:

```
const app = require('./app') // the actual Express application
const http = require('http')
const config = require('./utils/config')
const logger = require('./utils/logger')

const server = http.createServer(app)

server.listen(config.PORT, () => {
  logger.info(`Server running on port ${config.PORT}`)
})
```

The *index.js* file only imports the actual application from the *app.js* file and then starts the application. The function *info* of the logger-module is used for the console printout telling that the application is running.

The handling of environment variables is extracted into a separate *utils/config.js* file:

```
require('dotenv').config()

const PORT = process.env.PORT
const MONGODB_URI = process.env.MONGODB_URI

module.exports = {
```

```
MONGODB_URI,  
  PORT  
}
```

The other parts of the application can access the environment variables by importing the configuration module:

```
const config = require('./utils/config')  
  
logger.info(`Server running on port ${config.PORT}`)
```

The route handlers have also been moved into a dedicated module. The event handlers of routes are commonly referred to as *controllers*, and for this reason we have created a new *controllers* directory. All of the routes related to notes are now in the *notes.js* module under the *controllers* directory.

The contents of the *notes.js* module are the following:

```
const notesRouter = require('express').Router()
const Note = require('../models/note')

notesRouter.get('/', (request, response) => {
  Note.find({}).then(notes => {
    response.json(notes)
  })
})

notesRouter.get('/:id', (request, response, next) => {
  Note.findById(request.params.id)
    .then(note => {
      if (note) {
        response.json(note)
      } else {
        response.status(404).end()
      }
    })
    .catch(error => next(error))
})

notesRouter.post('/', (request, response, next) => {
  const body = request.body

  const note = new Note({
    content: body.content,
    important: body.important || false,
    date: new Date()
  })

  note.save()
    .then(savedNote => {
      response.json(savedNote)
    })
    .catch(error => next(error))
})

notesRouter.delete('/:id', (request, response, next) => {
  Note.findByIdAndRemove(request.params.id)
    .then(() => {
      response.status(204).end()
    })
    .catch(error => next(error))
})

notesRouter.put('/:id', (request, response, next) => {
  const body = request.body

  const note = {
    content: body.content,
    important: body.important,
  }
}
```

```
Note.findByIdAndUpdate(request.params.id, note, { new: true })
  .then(updatedNote => {
    response.json(updatedNote)
  })
  .catch(error => next(error))
})

module.exports = notesRouter
```

This is almost an exact copy-paste of our previous `index.js` file.

However, there are a few significant changes. At the very beginning of the file we create a new router object:

```
const notesRouter = require('express').Router()

//...

module.exports = notesRouter
```

The module exports the router to be available for all consumers of the module.

All routes are now defined for the router object, in a similar fashion to what we had previously done with the object representing the entire application.

It's worth noting that the paths in the route handlers have shortened. In the previous version, we had:

```
app.delete('/api/notes/:id', (request, response) => {
```

And in the current version, we have:

```
notesRouter.delete('/:id', (request, response) => {
```

So what are these router objects exactly? The Express manual provides the following explanation:

A router object is an isolated instance of middleware and routes. You can think of it as a “mini-application,” capable only of performing middleware and routing functions. Every Express application has a built-in app router.

The router is in fact a *middleware*, that can be used for defining "related routes" in a single place, that is typically placed in its own module.

The `app.js` file that creates the actual application, takes the router into use as shown below:

```
const notesRouter = require('./controllers/notes')
app.use('/api/notes', notesRouter)
```

The router we defined earlier is used *if* the URL of the request starts with `/api/notes`. For this reason, the `notesRouter` object must only define the relative parts of the routes, i.e. the empty path `/` or just the parameter `:id`.

After making these changes, our `app.js` file looks like this:

```
const config = require('./utils/config')
const express = require('express')
const app = express()
const cors = require('cors')
const notesRouter = require('./controllers/notes')
const middleware = require('./utils/middleware')
const logger = require('./utils/logger')
const mongoose = require('mongoose')

logger.info('connecting to', config.MONGODB_URI)

mongoose.connect(config.MONGODB_URI, { useNewUrlParser: true, useUnifiedTopology: true, useFindAndModify: false })
  .then(() => {
    logger.info('connected to MongoDB')
  })
  .catch((error) => {
    logger.error('error connecting to MongoDB:', error.message)
  })

app.use(cors())
app.use(express.static('build'))
app.use(express.json())
app.use(middleware.requestLogger)

app.use('/api/notes', notesRouter)

app.use(middleware.unknownEndpoint)
app.use(middleware.errorHandler)

module.exports = app
```

The file takes different middleware into use, and one of these is the `notesRouter` that is attached to the `/api/notes` route.

Our custom middleware has been moved to a new `utils/middleware.js` module:

```
const logger = require('./logger')

const requestLogger = (request, response, next) => {
  logger.info('Method:', request.method)
  logger.info('Path:  ', request.path)
  logger.info('Body:  ', request.body)
  logger.info('---')
  next()
}

const unknownEndpoint = (request, response) => {
  response.status(404).send({ error: 'unknown endpoint' })
}

const errorHandler = (error, request, response, next) => {
  logger.error(error.message)

  if (error.name === 'CastError') {
    return response.status(400).send({ error: 'malformatted id' })
  } else if (error.name === 'ValidationError') {
    return response.status(400).json({ error: error.message })
  }

  next(error)
}

module.exports = {
  requestLogger,
  unknownEndpoint,
  errorHandler
}
```

The responsibility of establishing the connection to the database has been given to the *app.js* module. The *note.js* file under the *models* directory only defines the Mongoose schema for notes.

```
const mongoose = require('mongoose')

const noteSchema = new mongoose.Schema({
  content: {
    type: String,
    required: true,
    minlength: 5
  },
  date: {
    type: Date,
    required: true,
  },
  important: Boolean,
})
```

```
noteSchema.set('toJSON', {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString()
    delete returnedObject._id
    delete returnedObject.__v
  }
})

module.exports = mongoose.model('Note', noteSchema)
```

To recap, the directory structure looks like this after the changes have been made:

```
├─ index.js
├─ app.js
├─ build
│   └─ ...
├─ controllers
│   └─ notes.js
├─ models
│   └─ note.js
├─ package-lock.json
├─ package.json
├─ utils
│   ├── config.js
│   ├── logger.js
│   └─ middleware.js
```

For smaller applications the structure does not matter that much. Once the application starts to grow in size, you are going to have to establish some kind of structure, and separate the different responsibilities of the application into separate modules. This will make developing the application much easier.

There is no strict directory structure or file naming convention that is required for Express applications. To contrast this, Ruby on Rails does require a specific structure. Our current structure simply follows some of the best practices you can come across on the internet.

You can find the code for our current application in its entirety in the *part4-1* branch of [this Github repository](#).

If you clone the project for yourself, run the `npm install` command before starting the application with `npm start`.

Exercises 4.1.-4.2.

In the exercises for this part we will be building a *blog list application*, that allows users to save

information about interesting blogs they have stumbled across on the internet. For each listed blog we will save the author, title, url, and amount of upvotes from users of the application.

4.1 Blog list, step1

Let's imagine a situation, where you receive an email that contains the following application body:

```
const http = require('http')
const express = require('express')
const app = express()
const cors = require('cors')
const mongoose = require('mongoose')

const blogSchema = new mongoose.Schema({
  title: String,
  author: String,
  url: String,
  likes: Number
})

const Blog = mongoose.model('Blog', blogSchema)

const mongoUrl = 'mongodb://localhost/bloglist'
mongoose.connect(mongoUrl, { useNewUrlParser: true, useUnifiedTopology: true, useFindAndModify: false,

app.use(cors())
app.use(express.json())

app.get('/api/blogs', (request, response) => {
  Blog
    .find({})
    .then(blogs => {
      response.json(blogs)
    })
})

app.post('/api/blogs', (request, response) => {
  const blog = new Blog(request.body)

  blog
    .save()
    .then(result => {
      response.status(201).json(result)
    })
})

const PORT = 3003
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`)
})
```

Turn the application into a functioning *npm* project. In order to keep your development productive, configure the application to be executed with *nodemon*. You can create a new database for your application with MongoDB Atlas, or use the same database from the previous part's exercises.

Verify that it is possible to add blogs to list with Postman or the VS Code REST client and that the application returns the added blogs at the correct endpoint.

4.2 Blog list, step2

Refactor the application into separate modules as shown earlier in this part of the course material.

NB refactor your application in baby steps and verify that the application works after every change you make. If you try to take a "shortcut" by refactoring many things at once, then Murphy's law will kick in and it is almost certain that something will break in your application. The "shortcut" will end up taking more time than moving forward slowly and systematically.

One best practice is to commit your code every time it is in a stable state. This makes it easy to rollback to a situation where the application still works.

Testing Node applications

We have completely neglected one essential area of software development, and that is automated testing.

Let's start our testing journey by looking at unit tests. The logic of our application is so simple, that there is not much that makes sense to test with unit tests. Let's create a new file *utils/for_testing.js* and write a couple of simple functions that we can use for test writing practice:

```
const palindrome = (string) => {
  return string
    .split('')
    .reverse()
    .join('')
}

const average = (array) => {
  const reducer = (sum, item) => {
    return sum + item
  }

  return array.reduce(reducer, 0) / array.length
}

module.exports = {
  palindrome,
  average,
}
```

The `average` function uses the array `reduce` method. If the method is not familiar to you yet, then now is a good time to watch the first three videos from the [Functional Javascript](#) series on Youtube.

There are many different testing libraries or *test runners* available for JavaScript. In this course we will be using a testing library developed and used internally by Facebook called `jest`, that resembles the previous king of JavaScript testing libraries `Mocha`.

Jest is a natural choice for this course, as it works well for testing backends, and it shines when it comes to testing React applications.

Windows users: Jest may not work if the path of the project directory contains a directory that has spaces in its name.

Since tests are only executed during the development of our application, we will install `jest` as a development dependency with the command:

```
npm install --save-dev jest
```

Let's define the `npm script test` to execute tests with Jest and to report about the test execution with the `verbose` style:

```
{
  //...
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "build:ui": "rm -rf build && cd ../../../../luento/notes && npm run build && cp -r build ../../../../3,
    "deploy": "git push heroku master",
    "deploy:full": "npm run build:ui && git add . && git commit -m uibuild && git push && npm run depl",
    "logs:prod": "heroku logs --tail",
    "lint": "eslint .",
    "test": "jest --verbose"
  },
  //...
}
```

Jest requires one to specify that the execution environment is Node. This can be done by adding the following to the end of `package.json`:

```
{
  //...
  "jest": {
    "testEnvironment": "node"
  }
}
```

```
}  
}
```

Alternatively, Jest can look for a configuration file with the default name *jest.config.js*, where we can define the execution environment like this:

```
module.exports = {  
  testEnvironment: 'node',  
};
```

Let's create a separate directory for our tests called *tests* and create a new file called *palindrome.test.js* with the following contents:

```
const palindrome = require('../utils/for_testing').palindrome  
  
test('palindrome of a', () => {  
  const result = palindrome('a')  
  
  expect(result).toBe('a')  
})  
  
test('palindrome of react', () => {  
  const result = palindrome('react')  
  
  expect(result).toBe('tcaer')  
})  
  
test('palindrome of releveler', () => {  
  const result = palindrome('releveler')  
  
  expect(result).toBe('releveler')  
})
```

The ESLint configuration we added to the project in the previous part complains about the `test` and `expect` commands in our test file, since the configuration does not allow *globals*. Let's get rid of the complaints by adding *"jest": true* to the *env* property in the *.eslintrc.js* file.

```
module.exports = {  
  'env': {  
    'commonjs': true,  
    'es2021': true,  
    'node': true,  
    'jest': true,  
  },
```

```
'extends': 'eslint:recommended',
'parserOptions': {
  'ecmaVersion': 12
},
'rules': {
  // ...
},
}
```

In the first row, the test file imports the function to be tested and assigns it to a variable called `palindrome` :

```
const palindrome = require('../utils/for_testing').palindrome
```

Individual test cases are defined with the `test` function. The first parameter of the function is the test description as a string. The second parameter is a *function*, that defines the functionality for the test case. The functionality for the second test case looks like this:

```
() => {
  const result = palindrome('react')

  expect(result).toBe('tcaer')
}
```

First we execute the code to be tested, meaning that we generate a palindrome for the string *react*. Next we verify the results with the `expect` function. `Expect` wraps the resulting value into an object that offers a collection of *matcher* functions, that can be used for verifying the correctness of the result. Since in this test case we are comparing two strings, we can use the `toBe` matcher.

As expected, all of the tests pass:



```
→ notes-backend git:(part4-2) x npm test
> hello@1.0.0 test /Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend
> jest --verbose

PASS tests/palindrome.test.js
  ✓ palindrom of a (5ms)
  ✓ palindrom of react (1ms)
  ✓ palindrom of saippukauppias (1ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.091s
Ran all test suites.
```

Jest expects by default that the names of test files contain `.test`. In this course, we will follow the

convention of naming our tests files with the extension `.test.js`.

Jest has excellent error messages, let's break the test to demonstrate this:

```
test('palindrom of react', () => {  
  const result = palindrome('react')  
  
  expect(result).toBe('tkaer')  
})
```

Running the tests above results in the following error message:



```
✓ palindrome of relever (1ms)  
  
• palindrome of react  
  
expect(received).toBe(expected) // Object.is equality  
  
Expected: "tkaer"  
Received: "tcaer"  
  
   10 |   const result = palindrome('react')  
   11 |  
>  12 |   expect(result).toBe('tkaer')  
      |                   ^  
   13 | })  
   14 |  
   15 | test('palindrome of relever', () => {  
  
    at Object.toBe (tests/palindrome.test.js:12:18)  
  
Test Suites: 1 failed, 1 total  
Tests:       1 failed, 2 passed, 3 total  
Snapshots:   0 total  
Time:        1.129s
```

Let's add a few tests for the `average` function, into a new file `tests/average.test.js`.

```

const average = require('../utils/for_testing').average

describe('average', () => {
  test('of one value is the value itself', () => {
    expect(average([1])).toBe(1)
  })

  test('of many is calculated right', () => {
    expect(average([1, 2, 3, 4, 5, 6])).toBe(3.5)
  })

  test('of empty array is zero', () => {
    expect(average([])).toBe(0)
  })
})

```

The test reveals that the function does not work correctly with an empty array (this is because in JavaScript dividing by zero results in *NaN*):

```

FAIL tests/average.test.js
average
  ✓ of one value is the value itself (2ms)
  ✓ of many is calculated right (1ms)
  ✗ of empty array is zero (10ms)

  • average › of empty array is zero

    expect(received).toBe(expected) // Object.is equality

    Expected: 0
    Received: NaN

    11 |
    12 |   test('of empty array is zero', () => {
  > 13 |     expect(average([])).toBe(0)
      |                   ^
    14 |   })
    15 | })

    at Object.toBe (tests/average.test.js:13:25)

Test Suites: 1 failed, 1 passed, 2 total
Tests:       1 failed, 5 passed, 6 total
Snapshots:   0 total
Time:        0.736s, estimated 1s
Ran all test suites.

```

Fixing the function is quite easy:

```

const average = array => {
  const reducer = (sum, item) => {
    return sum + item
  }

  return array.length === 0
    ? 0

```

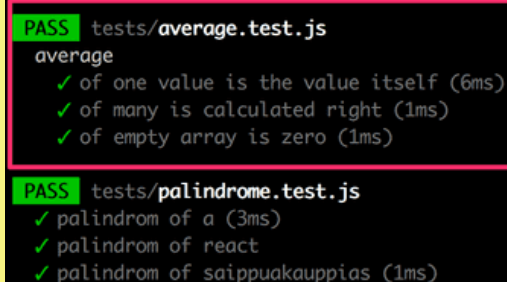
```
    : array.reduce(reducer, 0) / array.length  
  }  
}
```

If the length of the array is 0 then we return 0, and in all other cases we use the `reduce` method to calculate the average.

There are a few things to notice about the tests that we just wrote. We defined a *describe* block around the tests that was given the name `average` :

```
describe('average', () => {  
  // tests  
})
```

Describe blocks can be used for grouping tests into logical collections. The test output of Jest also uses the name of the describe block:



```
PASS tests/average.test.js  
  average  
    ✓ of one value is the value itself (6ms)  
    ✓ of many is calculated right (1ms)  
    ✓ of empty array is zero (1ms)  
  
PASS tests/palindrome.test.js  
  ✓ palindrom of a (3ms)  
  ✓ palindrom of react  
  ✓ palindrom of saippukauppias (1ms)
```

As we will see later on *describe* blocks are necessary when we want to run some shared setup or teardown operations for a group of tests.

Another thing to notice is that we wrote the tests in quite a compact way, without assigning the output of the function being tested to a variable:

```
test('of empty array is zero', () => {  
  expect(average([])).toBe(0)  
})
```

Exercises 4.3.-4.7.

Let's create a collection of helper functions that are meant to assist dealing with the blog list. Create the functions into a file called *utils/list_helper.js*. Write your tests into an appropriately named test file under the *tests* directory.

4.3: helper functions and unit tests, step1

First define a `dummy` function that receives an array of blog posts as a parameter and always returns the value 1. The contents of the `list_helper.js` file at this point should be the following:

```
const dummy = (blogs) => {  
  // ...  
}  
  
module.exports = {  
  dummy  
}
```

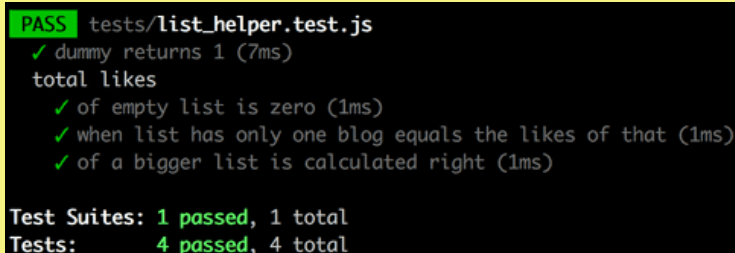
Verify that your test configuration works with the following test:

```
const listHelper = require('../utils/list_helper')  
  
test('dummy returns one', () => {  
  const blogs = []  
  
  const result = listHelper.dummy(blogs)  
  expect(result).toBe(1)  
})
```

4.4: helper functions and unit tests, step2

Define a new `totalLikes` function that receives a list of blog posts as a parameter. The function returns the total sum of *likes* in all of the blog posts.

Write appropriate tests for the function. It's recommended to put the tests inside of a *describe* block, so that the test report output gets grouped nicely:



```
PASS tests/list_helper.test.js  
✓ dummy returns 1 (7ms)  
total likes  
  ✓ of empty list is zero (1ms)  
  ✓ when list has only one blog equals the likes of that (1ms)  
  ✓ of a bigger list is calculated right (1ms)  
  
Test Suites: 1 passed, 1 total  
Tests:      4 passed, 4 total
```

Defining test inputs for the function can be done like this:

```
describe('total likes', () => {  
  const listWithOneBlog = [  
    {  

```

```

    _id: '5a422aa71b54a676234d17f8',
    title: 'Go To Statement Considered Harmful',
    author: 'Edsger W. Dijkstra',
    url: 'http://www.u.arizona.edu/~rubinson/copyright_violations/Go_To_Considered_Harmful.html',
    likes: 5,
    __v: 0
  }
]

test('when list has only one blog, equals the likes of that', () => {
  const result = listHelper.totalLikes(listWithOneBlog)
  expect(result).toBe(5)
})
})

```

If defining your own test input list of blogs is too much work, you can use the ready-made list [here](#).

You are [bound](#) to run into problems while writing tests. Remember the things that we learned about [debugging](#) in part 3. You can print things to the console with `console.log` even during test execution. It is even possible to use the debugger while running tests, you can find instructions for that [here](#).

NB: if some test is failing, then it is recommended to only run that test while you are fixing the issue. You can run a single test with the [only](#) method.

Another way of running a single test (or describe block) is to specify the name of the test to be run with the [-t](#) flag:

```
npm test -- -t 'when list has only one blog, equals the likes of that'
```

4.5*: helper functions and unit tests, step3

Define a new `favoriteBlog` function that receives a list of blogs as a parameter. The function finds out which blog has most likes. If there are many top favorites, it is enough to return one of them.

The value returned by the function could be in the following format:

```

{
  title: "Canonical string reduction",
  author: "Edsger W. Dijkstra",
  likes: 12
}

```

NB when you are comparing objects, the [toEqual](#) method is probably what you want to use, since

the `toBe` tries to verify that the two values are the same value, and not just that they contain the same properties.

Write the tests for this exercise inside of a new `describe` block. Do the same for the remaining exercises as well.

4.6*: helper functions and unit tests, step4

About course

This and the next exercise are a little bit more challenging. Finishing these two exercises is not required in order to advance in the course material, so it may be a good idea to return to these

Course contents once you're done going through the material for this part in its entirety.

Finishing this exercise can be done without the use of additional libraries. However, this exercise is a great opportunity to learn how to use the Lodash library.

Define a function called `mostBlogs` that receives an array of blogs as a parameter. The function returns the *author* who has the largest amount of blogs. The return value also contains the number of blogs the top author has:

Challenge

```
{
  author: "Robert C. Martin",
  blogs: 3
}
```

If there are many top bloggers, then it is enough to return any one of them.

4.7*: helper functions and unit tests, step5

Define a function called `mostLikes` that receives an array of blogs as its parameter. The function returns the *author* whose blog posts have the largest amount of likes. The return value also contains the total amount of likes that the author has received:

```
{
  author: "Edsger W. Dijkstra",
  likes: 17
}
```

If there are many top bloggers, then it is enough to show any one of them.

[Propose changes to material](#)

Part 3
Previous part

Part 4b
Next part

