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

Note: this course material was written with version v20.11.0 of Node.js. Please make sure that your version of Node is at least as new as the version used in the material (you can check the version by running `node -v` in the command line).

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

Once we make the changes to the directory structure of our project, we will end up with the following structure:

```
├── index.js
├── app.js
├── dist
|   └── ...
├── controllers
|   └── notes.js
├── models
|   └── note.js
├── package-lock.json
├── package.json
```

copy

```
├── 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 several ways. 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 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 contents of the *index.js* file used for starting the application gets simplified as follows:

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

app.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.

Now the Express app and the code taking care of the web server are separated from each other following the best practices. One of the advantages of this method is that the application can now be tested at the level of HTTP API calls without actually making calls via HTTP over the network, this makes the execution of tests faster.

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,
  })

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

notesRouter.delete('/:id', (request, response, next) => {
  Note.findByIdAndDelete(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, similar to what was done before 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, next) => {
```

copy

And in the current version, we have:

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

copy

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, which 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)
```

copy

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')

mongoose.set('strictQuery', false)

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

copy

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

app.use(cors())
app.use(express.static('dist'))
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
  },
  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
├── dist
|   └── ...
├── 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. In contrast, Ruby on Rails does require a specific structure. Our current structure simply follows some of the best practices that 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 run dev`.

Note on exports

We have used two different kinds of exports in this part. Firstly, e.g. the file *utils/logger.js* does the export as follows:

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

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

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

The file exports *an object* that has two fields, both of which are functions. The functions can be used in two different ways. The first option is to require the whole object and refer to functions through the object using the dot notation:

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

logger.info('message')

logger.error('error message')
```

The other option is to destructure the functions to their own variables in the *require* statement:

```
const { info, error } = require('./utils/logger')
```

```
info('message')
error('error message')
```

The second way of exporting may be preferable if only a small portion of the exported functions are used in a file. E.g. in file *controller/notes.js* exporting happens as follows:

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

// ...

module.exports = notesRouter
```

In this case, there is just one "thing" exported, so the only way to use it is the following:

```
const notesRouter = require('./controllers/notes')

// ...

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

Now the exported "thing" (in this case a router object) is assigned to a variable and used as such.

Finding the usages of your exports with VS Code

VS Code has a handy feature that allows you to see where your modules have been exported. This can be very helpful for refactoring. For example, if you decide to split a function into two separate functions, your code could break if you don't modify all the usages. This is difficult if you don't know where they are. However, you need to define your exports in a particular way for this to work.

If you right-click on a variable in the location it is exported from and select "Find All References", it will show you everywhere the variable is imported. However, if you assign an object directly to module.exports, it will not work. A workaround is to assign the object you want to export to a named variable and then export the named variable. It also will not work if you destructure where you are importing; you have to import the named variable and then destructure, or just use dot notation to use the functions contained in the named variable.

The nature of VS Code bleeding into how you write your code is probably not ideal, so you need to decide for yourself if the trade-off is worthwhile.

Exercises 4.1.-4.2.

Note: this course material was written with version v20.11.0 of Node.js. Please make sure that your version of Node is at least as new as the version used in the material (you can check the version by running `node -v` in the command line).

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, step 1

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

```
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)

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. 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 the 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, step 2

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 it 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.

If you're having issues with *content.body* being *undefined* for seemingly no reason, make sure you didn't forget to add *app.use(express.json())* near the top of the file.

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 reverse = (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 = {
  reverse,
  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 a large number of test libraries, or *test runners*, available for JavaScript. The old king of test libraries is Mocha, which was replaced a few years ago by Jest. A newcomer to the libraries is Vitest, which bills itself as a new generation of test libraries.

Nowadays, Node also has a built-in test library node:test, which is well suited to the needs of the course.

Let's define the *npm script* `test` for the test execution:

copy

```
{
  //...
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "build:ui": "rm -rf build && cd ../frontend/ && npm run build && cp -r build
../backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs",
    "lint": "eslint .",
    "test": "node --test"
  },
  //...
}
```

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

copy

```
const { test } = require('node:test')
const assert = require('node:assert')

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

test('reverse of a', () => {
  const result = reverse('a')

  assert.strictEqual(result, 'a')
})
```

```
test('reverse of react', () => {
  const result = reverse('react')

  assert.strictEqual(result, 'tcaer')
})

test('reverse of saippuakauppias', () => {
  const result = reverse('saippuakauppias')

  assert.strictEqual(result, 'saippuakauppias')
})
```

The test defines the keyword `test` and the library assert, which is used by the tests to check the results of the functions under test.

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

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

copy

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

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

  assert.strictEqual(result, 'tcaer')
}
```

copy

First, we execute the code to be tested, meaning that we generate a reverse for the string *react*. Next, we verify the results with the the method strictEqual of the assert library.

As expected, all of the tests pass:

```
→  notes-backend git:(part4-2) npm test

> notebackend@1.0.0 test
> node --test

✔ reverse of a (0.443708ms)
✔ reverse of react (0.1045ms)
✔ reverse of saippuakauppias (0.045ms)
ℹ tests 3
ℹ suites 0
ℹ pass 3
ℹ fail 0
ℹ cancelled 0
ℹ skipped 0
ℹ todo 0
ℹ duration_ms 55.315667
```

The library node:test 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*.

Let's break the test:

```
test('reverse of react', () => {
  const result = reverse('react')

  assert.strictEqual(result, 'tkaer')
})
```

Running this test results in the following error message:

```
✖ failing tests:

test at tests/reverse.test.js:12:1
✖ reverse of react (0.781584ms)
  AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
  + actual - expected

  + 'tcaer'
  - 'tkaer'
```

Let's put the tests for the `average` function, into a new file called *tests/average.test.js*.

```
const { test, describe } = require('node:test')

// ...

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

describe('average', () => {
```

```
  test('of one value is the value itself', () => {
    assert.strictEqual(average([1]), 1)
  })

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

  test('of empty array is zero', () => {
    assert.strictEqual(average([]), 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*):

```
    generatedMessage: true,
    code: 'ERR_ASSERTION',
    actual: 'tcaer',
    expected: 'tkaer',
    operator: 'strictEqual'
  }

test at tests/reverse.test.js:34:3
✖ of empty array is zero (0.1025ms)
  AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:

  NaN !== 0
```

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
}
```

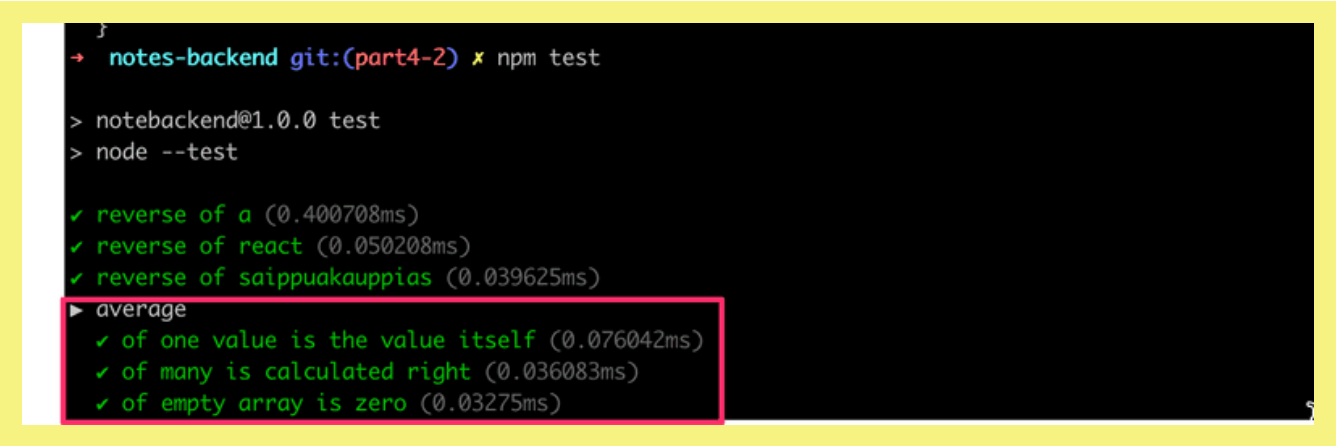
If the length of the array is 0 then we return 0, and in all other cases, we use the `reduce` method to calculate the average.

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

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

```
})
```

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



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', () => {
  assert.strictEqual(average([]), 0)
})
```

Exercises 4.3.-4.7.

Let's create a collection of helper functions that are best suited for working with the describe sections of 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, step 1

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 { test, describe } = require('node:test')
const assert = require('node:assert')
const listHelper = require('../utils/list_helper')

test('dummy returns one', () => {
  const blogs = []

  const result = listHelper.dummy(blogs)
  assert.strictEqual(result, 1)
})
```

4.4: Helper Functions and Unit Tests, step 2

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: 'https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf',
      likes: 5,
      __v: 0
    }
  ]

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

```
    assert.strictEqual(result, 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.

4.5*: Helper Functions and Unit Tests, step 3

Define a new `favoriteBlog` function that receives a list of blogs as a parameter. The function finds out which blog has the 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 deepStrictEqual method is probably what you want to use, since the strictEqual tries to verify that the two values are the same value, and not just that they contain the same properties. For differences between various assert module functions, you can refer to this Stack Overflow answer.

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, step 4

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

```
{
  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, step 5

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 number 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

About course

Course contents

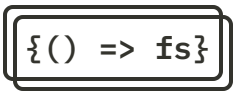
FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON



b Testing the backend

We will now start writing tests for the backend. Since the backend does not contain any complicated logic, it doesn't make sense to write unit tests for it. The only potential thing we could unit test is the `toJSON` method that is used for formatting notes.

In some situations, it can be beneficial to implement some of the backend tests by mocking the database instead of using a real database. One library that could be used for this is mongodb-memory-server.

Since our application's backend is still relatively simple, we will decide to test the entire application through its REST API, so that the database is also included. This kind of testing where multiple components of the system are being tested as a group is called integration testing.

Test environment

In one of the previous chapters of the course material, we mentioned that when your backend server is running in Fly.io or Render, it is in *production* mode.

The convention in Node is to define the execution mode of the application with the *NODE_ENV* environment variable. In our current application, we only load the environment variables defined in the *.env* file if the application is *not* in production mode.

It is common practice to define separate modes for development and testing.

Next, let's change the scripts in our notes application *package.json* file, so that when tests are run, *NODE_ENV* gets the value *test*:

```
{
  // ...
  "scripts": {
    "start": "NODE_ENV=production node index.js",
    "dev": "NODE_ENV=development nodemon index.js",
    "test": "NODE_ENV=test node --test",
    "build:ui": "rm -rf build && cd ../frontend/ && npm run build && cp -r build
../backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs",
    "lint": "eslint .",
  },
  // ...
}
```

We specified the mode of the application to be *development* in the `npm run dev` script that uses nodemon. We also specified that the default `npm start` command will define the mode as *production*.

There is a slight issue in the way that we have specified the mode of the application in our scripts: it will not work on Windows. We can correct this by installing the cross-env package as a development dependency with the command:

```
npm install --save-dev cross-env
```

We can then achieve cross-platform compatibility by using the cross-env library in our npm scripts defined in *package.json*:

```
{
  // ...
  "scripts": {
    "start": "cross-env NODE_ENV=production node index.js",
    "dev": "cross-env NODE_ENV=development nodemon index.js",
    "test": "cross-env  NODE_ENV=test node --test",
    // ...
  },
  // ...
}
```

NB: If you are deploying this application to Fly.io/Render, keep in mind that if cross-env is saved as a development dependency, it would cause an application error on your web server. To fix this, change cross-env to a production dependency by running this in the command line:

```
npm install cross-env
```

Now we can modify the way that our application runs in different modes. As an example of this, we could define the application to use a separate test database when it is running tests.

We can create our separate test database in MongoDB Atlas. This is not an optimal solution in situations where many people are developing the same application. Test execution in particular typically requires a single database instance that is not used by tests that are running concurrently.

It would be better to run our tests using a database that is installed and running on the developer's local machine. The optimal solution would be to have every test execution use a separate database. This is "relatively simple" to achieve by running Mongo in-memory or by using Docker containers. We will not complicate things and will instead continue to use the MongoDB Atlas database.

Let's make some changes to the module that defines the application's configuration in `utils/config.js`:

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

const PORT = process.env.PORT

const MONGODB_URI = process.env.NODE_ENV === 'test'
  ? process.env.TEST_MONGODB_URI
  : process.env.MONGODB_URI

module.exports = {
  MONGODB_URI,
  PORT
}
```

The *.env* file has *separate variables* for the database addresses of the development and test databases:

```
MONGODB_URI=mongodb+srv://fullstack:thepasswordishere@cluster0.o1opl.mongodb.net/noteApp?
retryWrites=true&w=majority
PORT=3001

TEST_MONGODB_URI=mongodb+srv://fullstack:thepasswordishere@cluster0.o1opl.mongodb.net/testNo
retryWrites=true&w=majority
```

The `config` module that we have implemented slightly resembles the node-config package. Writing our implementation is justified since our application is simple, and also because it teaches us valuable lessons.

These are the only changes we need to make to our application's code.

You can find the code for our current application in its entirety in the *part4-2* branch of this GitHub repository.

supertest

Let's use the supertest package to help us write our tests for testing the API.

We will install the package as a development dependency:

```
npm install --save-dev supertest
```

copy

Let's write our first test in the *tests/note_api.test.js* file:

```
const { test, after } = require('node:test')
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')

const api = supertest(app)

test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
})

after(async () => {
  await mongoose.connection.close()
})
```

copy

The test imports the Express application from the *app.js* module and wraps it with the *supertest* function into a so-called superagent object. This object is assigned to the *api* variable and tests can use it for making HTTP requests to the backend.

Our test makes an HTTP GET request to the *api/notes* url and verifies that the request is responded to with the status code 200. The test also verifies that the *Content-Type* header is set to *application/json*, indicating that the data is in the desired format.

Checking the value of the header uses a bit strange looking syntax:

```
.expect('Content-Type', /application\/json/)
```

copy

The desired value is now defined as regular expression or in short regex. The regex starts and ends with a slash /, because the desired string *application/json* also contains the same slash, it is preceded by a \ so that it is not interpreted as a regex termination character.

In principle, the test could also have been defined as a string

```
.expect('Content-Type', 'application/json')
```

copy

The problem here, however, is that when using a string, the value of the header must be exactly the same. For the regex we defined, it is acceptable that the header *contains* the string in question. The actual value of the header is *application/json; charset=utf-8*, i.e. it also contains information about character encoding. However, our test is not interested in this and therefore it is better to define the test as a regex instead of an exact string.

The test contains some details that we will explore a bit later on. The arrow function that defines the test is preceded by the *async* keyword and the method call for the *api* object is preceded by the *await* keyword. We will write a few tests and then take a closer look at this async/await magic. Do not concern yourself with them for now, just be assured that the example tests work correctly. The async/await syntax is related to the fact that making a request to the API is an *asynchronous* operation. The async/await syntax can be used for writing asynchronous code with the appearance of synchronous code.

Once all the tests (there is currently only one) have finished running we have to close the database connection used by Mongoose. This can be easily achieved with the after method:

```
after(async () => {
  await mongoose.connection.close()
})
```

copy

One tiny but important detail: at the beginning of this part we extracted the Express application into the *app.js* file, and the role of the *index.js* file was changed to launch the application at the specified port via `app.listen`:

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

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

copy

The tests only use the Express application defined in the *app.js* file, which does not listen to any ports:

```
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')

const api = supertest(app)

// ...
```

The documentation for supertest says the following:

> *if the server is not already listening for connections then it is bound to an ephemeral port for you so there is no need to keep track of ports.*

In other words, supertest takes care that the application being tested is started at the port that it uses internally.

Let's add two notes to the test database using the `mongo.js` program (here we must remember to switch to the correct database url).

Let's write a few more tests:

```
test('there are two notes', async () => {
  const response = await api.get('/api/notes')

  assert.strictEqual(response.body.length, 2)
})

test('the first note is about HTTP methods', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(e => e.content)
  assert.strictEqual(contents.includes('HTML is easy'), true)
})
```

Both tests store the response of the request to the `response` variable, and unlike the previous test that used the methods provided by `supertest` for verifying the status code and headers, this time we are inspecting the response data stored in *response.body* property. Our tests verify the format and content of the response data with the method strictEqual of the assert-library.

We could simplify the second test a bit, and use the assert itself to verify that the note is among the returned ones:

```
test('the first note is about HTTP methods', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(e => e.content)
  // is the argument truthy
```

```
  assert(contents.includes('HTML is easy'))
})
```

The benefit of using the async/await syntax is starting to become evident. Normally we would have to use callback functions to access the data returned by promises, but with the new syntax things are a lot more comfortable:

```
const response = await api.get('/api/notes')

// execution gets here only after the HTTP request is complete
// the result of HTTP request is saved in variable response
assert.strictEqual(response.body.length, 2)
```

The middleware that outputs information about the HTTP requests is obstructing the test execution output. Let us modify the logger so that it does not print to the console in test mode:

```
const info = (...params) => {
  if (process.env.NODE_ENV !== 'test') {
    console.log(...params)
  }
}

const error = (...params) => {
  if (process.env.NODE_ENV !== 'test') {
    console.error(...params)
  }
}

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

Initializing the database before tests

Testing appears to be easy and our tests are currently passing. However, our tests are bad as they are dependent on the state of the database, that now happens to have two notes. To make them more robust, we have to reset the database and generate the needed test data in a controlled manner before we run the tests.

Our tests are already using the `after` function of to close the connection to the database after the tests are finished executing. The library node:test offers many other functions that can be used for executing operations once before any test is run or every time before a test is run.

Let's initialize the database *before every test* with the `beforeEach` function:

```
const { test, after, beforeEach } = require('node:test')
```

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

const initialNotes = [
  {
    content: 'HTML is easy',
    important: false,
  },
  {
    content: 'Browser can execute only JavaScript',
    important: true,
  },
]

// ...

beforeEach(async () => {
  await Note.deleteMany({})
  let noteObject = new Note(initialNotes[0])
  await noteObject.save()
  noteObject = new Note(initialNotes[1])
  await noteObject.save()
})
// ...
```

The database is cleared out at the beginning, and after that, we save the two notes stored in the `initialNotes` array to the database. By doing this, we ensure that the database is in the same state before every test is run.

Let's also make the following changes to the last two tests:

```
test('there are two notes', async () => {
  const response = await api.get('/api/notes')

  assert.strictEqual(response.body.length, initialNotes.length)
})

test('the first note is about HTTP methods', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(e => e.content)
  assert(contents.includes('HTML is easy'))
})
```

Running tests one by one

The `npm test` command executes all of the tests for the application. When we are writing tests, it is usually wise to only execute one or two tests.

There are a few different ways of accomplishing this, one of which is the `only` method. With this method we can define in the code what tests should be executed:

```
test.only('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
})

test.only('there are two notes', async () => {
  const response = await api.get('/api/notes')

  assert.strictEqual(response.body.length, 2)
})
```

copy

When tests are run with option `--test-only`, that is, with the command:

```
npm test -- --test-only
```

copy

only the `only` marked tests are executed.

The danger of `only` is that one forgets to remove those from the code.

Another option is to specify the tests that need to be run as arguments of the *npm test* command.

The following command only runs the tests found in the *tests/note_api.test.js* file:

```
npm test -- tests/note_api.test.js
```

copy

The --tests-by-name-pattern option can be used for running tests with a specific name:

```
npm test -- --test-name-pattern="the first note is about HTTP methods"
```

copy

The provided argument can refer to the name of the test or the describe block. It can also contain just a part of the name. The following command will run all of the tests that contain *notes* in their name:

```
npm run test -- --test-name-pattern="notes"
```

copy

async/await

Before we write more tests let's take a look at the `async` and `await` keywords.

The async/await syntax that was introduced in ES7 makes it possible to use *asynchronous functions that return a promise* in a way that makes the code look synchronous.

As an example, the fetching of notes from the database with promises looks like this:

```
Note.find({}).then(notes => {
  console.log('operation returned the following notes', notes)
})
```

The `Note.find()` method returns a promise and we can access the result of the operation by registering a callback function with the `then` method.

All of the code we want to execute once the operation finishes is written in the callback function. If we wanted to make several asynchronous function calls in sequence, the situation would soon become painful. The asynchronous calls would have to be made in the callback. This would likely lead to complicated code and could potentially give birth to a so-called callback hell.

By chaining promises we could keep the situation somewhat under control, and avoid callback hell by creating a fairly clean chain of `then` method calls. We have seen a few of these during the course. To illustrate this, you can view an artificial example of a function that fetches all notes and then deletes the first one:

```
Note.find({})
  .then(notes => {
    return notes[0].deleteOne()
  })
  .then(response => {
    console.log('the first note is removed')
    // more code here
  })
```

The then-chain is alright, but we can do better. The generator functions introduced in ES6 provided a clever way of writing asynchronous code in a way that "looks synchronous". The syntax is a bit clunky and not widely used.

The `async` and `await` keywords introduced in ES7 bring the same functionality as the generators, but in an understandable and syntactically cleaner way to the hands of all citizens of the JavaScript world.

We could fetch all of the notes in the database by utilizing the await operator like this:

```
const notes = await Note.find({})

console.log('operation returned the following notes', notes)
```

The code looks exactly like synchronous code. The execution of code pauses at `const notes = await Note.find({})` and waits until the related promise is *fulfilled*, and then continues its execution to the next line. When the execution continues, the result of the operation that returned a promise is assigned to the `notes` variable.

The slightly complicated example presented above could be implemented by using await like this:

```
const notes = await Note.find({})
const response = await notes[0].deleteOne()

console.log('the first note is removed')
```

Thanks to the new syntax, the code is a lot simpler than the previous then-chain.

There are a few important details to pay attention to when using async/await syntax. To use the await operator with asynchronous operations, they have to return a promise. This is not a problem as such, as regular asynchronous functions using callbacks are easy to wrap around promises.

The await keyword can't be used just anywhere in JavaScript code. Using await is possible only inside of an async function.

This means that in order for the previous examples to work, they have to be using async functions. Notice the first line in the arrow function definition:

```
const main = async () => {
  const notes = await Note.find({})
  console.log('operation returned the following notes', notes)

  const response = await notes[0].deleteOne()
  console.log('the first note is removed')
}

main()
```

The code declares that the function assigned to `main` is asynchronous. After this, the code calls the function with `main()`.

async/await in the backend

Let's start to change the backend to async and await. As all of the asynchronous operations are currently done inside of a function, it is enough to change the route handler functions into async functions.

The route for fetching all notes gets changed to the following:

```
notesRouter.get('/', async (request, response) => {
  const notes = await Note.find({})
  response.json(notes)
})
```

We can verify that our refactoring was successful by testing the endpoint through the browser and by running the tests that we wrote earlier.

You can find the code for our current application in its entirety in the *part4-3* branch of this GitHub repository.

More tests and refactoring the backend

When code gets refactored, there is always the risk of regression, meaning that existing functionality may break. Let's refactor the remaining operations by first writing a test for each route of the API.

Let's start with the operation for adding a new note. Let's write a test that adds a new note and verifies that the number of notes returned by the API increases and that the newly added note is in the list.

```
test('a valid note can be added ', async () => {
  const newNote = {
    content: 'async/await simplifies making async calls',
    important: true,
  }

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(201)
    .expect('Content-Type', /application\/json/)

  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)

  assert.strictEqual(response.body.length, initialNotes.length + 1)

  assert(contents.includes('async/await simplifies making async calls'))
})
```

The test fails because we accidentally returned the status code *200 OK* when a new note is created. Let us change that to the status code *201 CREATED*:

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

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

  note.save()
    .then(savedNote => {
      response.status(201).json(savedNote)
    })
    .catch(error => next(error))
})
```

Let's also write a test that verifies that a note without content will not be saved into the database.

```
test('note without content is not added', async () => {
  const newNote = {
    important: true
  }

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(400)

  const response = await api.get('/api/notes')

  assert.strictEqual(response.body.length, initialNotes.length)
})
```

Both tests check the state stored in the database after the saving operation, by fetching all the notes of the application.

```
const response = await api.get('/api/notes')
```

The same verification steps will repeat in other tests later on, and it is a good idea to extract these steps into helper functions. Let's add the function into a new file called *tests/test_helper.js* which is in the same directory as the test file.

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

const initialNotes = [
  {
    content: 'HTML is easy',
    important: false
  },
  {
    content: 'Browser can execute only JavaScript',
```

```
    important: true
  }
]

const nonExistingId = async () => {
  const note = new Note({ content: 'willremovethissoon' })
  await note.save()
  await note.deleteOne()

  return note._id.toString()
}

const notesInDb = async () => {
  const notes = await Note.find({})
  return notes.map(note => note.toJSON())
}

module.exports = {
  initialNotes, nonExistingId, notesInDb
}
```

The module defines the `notesInDb` function that can be used for checking the notes stored in the database. The `initialNotes` array containing the initial database state is also in the module. We also define the `nonExistingId` function ahead of time, which can be used for creating a database object ID that does not belong to any note object in the database.

Our tests can now use the helper module and be changed like this:

copy

```
const { test, after, beforeEach } = require('node:test')
const assert = require('node:assert')
const supertest = require('supertest')
const mongoose = require('mongoose')
const helper = require('./test_helper')
const app = require('../app')
const api = supertest(app)

const Note = require('../models/note')

beforeEach(async () => {
  await Note.deleteMany({})

  let noteObject = new Note(helper.initialNotes[0])
  await noteObject.save()

  noteObject = new Note(helper.initialNotes[1])
  await noteObject.save()
})

test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
```

```
})

test('all notes are returned', async () => {
  const response = await api.get('/api/notes')

   assert.strictEqual(response.body.length, helper.initialNotes.length)
})

test('a specific note is within the returned notes', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)

  assert(contents.includes('Browser can execute only JavaScript'))
})

test('a valid note can be added ', async () => {
  const newNote = {
    content: 'async/await simplifies making async calls',
    important: true,
  }

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(201)
    .expect('Content-Type', /application\/json/)

  const notesAtEnd = await helper.notesInDb()
  assert.strictEqual(notesAtEnd.length, helper.initialNotes.length + 1)

  const contents = notesAtEnd.map(n => n.content)
  assert(contents.includes('async/await simplifies making async calls'))
})

test('note without content is not added', async () => {
  const newNote = {
    important: true
  }

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(400)

  const notesAtEnd = await helper.notesInDb()

  assert.strictEqual(notesAtEnd.length, helper.initialNotes.length)
})

after(async () => {
  await mongoose.connection.close()
})
```

The code using promises works and the tests pass. We are ready to refactor our code to use the async/await syntax.

We make the following changes to the code that takes care of adding a new note (notice that the route handler definition is preceded by the `async` keyword):

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

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

  const savedNote = await note.save()
  response.status(201).json(savedNote)
})
```

There's a slight problem with our code: we don't handle error situations. How should we deal with them?

Error handling and async/await

If there's an exception while handling the POST request we end up in a familiar situation:

```
Method: POST
Path:   /api/notes
Body:   { important: true }
---
(node:89372) UnhandledPromiseRejectionWarning: ValidationError: Note validation failed: content: Path `content` is
 required.
    at new ValidationError (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/
lib/error/validation.js:30:11)
    at model.Document.invalidate (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mon
goose/lib/document.js:2071:32)
    at p.doValidate.skipSchemaValidators (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_mod
ules/mongoose/lib/document.js:1934:17)
    at /Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/lib/schematype.js:929
:9
    at process._tickCallback (internal/process/next_tick.js:172:11)
(node:89372) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwi
```

In other words, we end up with an unhandled promise rejection, and the request never receives a response.

With async/await the recommended way of dealing with exceptions is the old and familiar `try/catch` mechanism:

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

```
  const note = new Note({
    content: body.content,
    important: body.important || false,
  })
  try {
    const savedNote = await note.save()
    response.status(201).json(savedNote)
  } catch(exception) {
    next(exception)
  }
})
```

The catch block simply calls the `next` function, which passes the request handling to the error handling middleware.

After making the change, all of our tests will pass once again.

Next, let's write tests for fetching and removing an individual note:

```
test('a specific note can be viewed', async () => {
  const notesAtStart = await helper.notesInDb()

  const noteToView = notesAtStart[0]

  const resultNote = await api
    .get(`/api/notes/${noteToView.id}`)
    .expect(200)
    .expect('Content-Type', /application\/json/)

  assert.deepStrictEqual(resultNote.body, noteToView)
})

test('a note can be deleted', async () => {
  const notesAtStart = await helper.notesInDb()
  const noteToDelete = notesAtStart[0]

  await api
    .delete(`/api/notes/${noteToDelete.id}`)
    .expect(204)

  const notesAtEnd = await helper.notesInDb()

  const contents = notesAtEnd.map(r => r.content)
  assert(!contents.includes(noteToDelete.content))

  assert.strictEqual(notesAtEnd.length, helper.initialNotes.length - 1)
})
```

Both tests share a similar structure. In the initialization phase, they fetch a note from the database. After this, the tests call the actual operation being tested, which is highlighted in the code block. Lastly, the tests verify that the outcome of the operation is as expected.

There is one point worth noting in the first test. Instead of the previously used method strictEqual, the method deepStrictEqual is used:

```
assert.deepStrictEqual(resultNote.body, noteToView)
```

The reason for this is that `strictEqual` uses the method Object.is to compare similarity, i.e. it compares whether the objects are the same. In our case, it is enough to check that the contents of the objects, i.e. the values of their fields, are the same. For this purpose `deepStrictEqual` is suitable.

The tests pass and we can safely refactor the tested routes to use async/await:

```
notesRouter.get('/:id', async (request, response, next) => {
  try {
    const note = await Note.findById(request.params.id)
    if (note) {
      response.json(note)
    } else {
      response.status(404).end()
    }
  } catch(exception) {
    next(exception)
  }
})

notesRouter.delete('/:id', async (request, response, next) => {
  try {
    await Note.findByIdAndDelete(request.params.id)
    response.status(204).end()
  } catch(exception) {
    next(exception)
  }
})
```

You can find the code for our current application in its entirety in the *part4-4* branch of this GitHub repository.

Eliminating the try-catch

Async/await unclutters the code a bit, but the 'price' is the *try/catch* structure required for catching exceptions. All of the route handlers follow the same structure

```
try {
  // do the async operations here
} catch(exception) {
```

```
    next(exception)
}
```

One starts to wonder if it would be possible to refactor the code to eliminate the *catch* from the methods?

The express-async-errors library has a solution for this.

Let's install the library

```
npm install express-async-errors
```

copy

Using the library is *very* easy. You introduce the library in *app.js*, `before` you import your routes:

```
const config = require('./utils/config')
const express = require('express')
require('express-async-errors')
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')

// ...

module.exports = app
```

copy

The 'magic' of the library allows us to eliminate the try-catch blocks completely. For example the route for deleting a note

```
notesRouter.delete('/:id', async (request, response, next) => {
  try {
    await Note.findByIdAndDelete(request.params.id)
    response.status(204).end()
  } catch (exception) {
    next(exception)
  }
})
```

copy

becomes

```
notesRouter.delete('/:id', async (request, response) => {
  await Note.findByIdAndDelete(request.params.id)
```

copy

```
  response.status(204).end()
})
```

Because of the library, we do not need the `next(exception)` call anymore. The library handles everything under the hood. If an exception occurs in an *async* route, the execution is automatically passed to the error-handling middleware.

The other routes become:

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

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

  const savedNote = await note.save()
  response.status(201).json(savedNote)
})

notesRouter.get('/:id', async (request, response) => {
  const note = await Note.findById(request.params.id)
  if (note) {
    response.json(note)
  } else {
    response.status(404).end()
  }
})
```

Optimizing the beforeEach function

Let's return to writing our tests and take a closer look at the `beforeEach` function that sets up the tests:

```
beforeEach(async () => {
  await Note.deleteMany({})

  let noteObject = new Note(helper.initialNotes[0])
  await noteObject.save()

  noteObject = new Note(helper.initialNotes[1])
  await noteObject.save()
})
```

The function saves the first two notes from the `helper.initialNotes` array into the database with two separate operations. The solution is alright, but there's a better way of saving multiple objects to the

database:

```
beforeEach(async () => {
  await Note.deleteMany({})
  console.log('cleared')

  helper.initialNotes.forEach(async (note) => {
    let noteObject = new Note(note)
    await noteObject.save()
    console.log('saved')
  })
  console.log('done')
})

test('notes are returned as json', async () => {
  console.log('entered test')
  // ...
}
```

We save the notes stored in the array into the database inside of a `forEach` loop. The tests don't quite seem to work however, so we have added some console logs to help us find the problem.

The console displays the following output:

```
cleared
done
entered test
saved
saved
```

Despite our use of the async/await syntax, our solution does not work as we expected it to. The test execution begins before the database is initialized!

The problem is that every iteration of the forEach loop generates an asynchronous operation, and `beforeEach` won't wait for them to finish executing. In other words, the `await` commands defined inside of the `forEach` loop are not in the `beforeEach` function, but in separate functions that `beforeEach` will not wait for.

Since the execution of tests begins immediately after `beforeEach` has finished executing, the execution of tests begins before the database state is initialized.

One way of fixing this is to wait for all of the asynchronous operations to finish executing with the `Promise.all` method:

```
beforeEach(async () => {
  await Note.deleteMany({})
```

```
  const noteObjects = helper.initialNotes
    .map(note => new Note(note))
  const promiseArray = noteObjects.map(note => note.save())
  await Promise.all(promiseArray)
})
```

The solution is quite advanced despite its compact appearance. The `noteObjects` variable is assigned to an array of Mongoose objects that are created with the `Note` constructor for each of the notes in the `helper.initialNotes` array. The next line of code creates a new array that *consists of promises*, that are created by calling the `save` method of each item in the `noteObjects` array. In other words, it is an array of promises for saving each of the items to the database.

The Promise.all method can be used for transforming an array of promises into a single promise, that will be *fulfilled* once every promise in the array passed to it as an argument is resolved. The last line of code `await Promise.all(promiseArray)` waits until every promise for saving a note is finished, meaning that the database has been initialized.

> The returned values of each promise in the array can still be accessed when using the Promise.all method. If we wait for the promises to be resolved with the `await` syntax `const results = await Promise.all(promiseArray)`, the operation will return an array that contains the resolved values for each promise in the `promiseArray`, and they appear in the same order as the promises in the array.

Promise.all executes the promises it receives in parallel. If the promises need to be executed in a particular order, this will be problematic. In situations like this, the operations can be executed inside of a for...of block, that guarantees a specific execution order.

```
beforeEach(async () => {
  await Note.deleteMany({})

  for (let note of helper.initialNotes) {
    let noteObject = new Note(note)
    await noteObject.save()
  }
})
```

The asynchronous nature of JavaScript can lead to surprising behavior, and for this reason, it is important to pay careful attention when using the async/await syntax. Even though the syntax makes it easier to deal with promises, it is still necessary to understand how promises work!

The code for our application can be found on GitHub, branch *part4-5*.

A true full stack developer's oath

Making tests brings yet another layer of challenge to programming. We have to update our full stack developer oath to remind you that systematicity is also key when developing tests.

So we should once more extend our oath:

Full stack development is *extremely hard*, that is why I will use all the possible means to make it easier

- I will have my browser developer console open all the time
- I will use the network tab of the browser dev tools to ensure that frontend and backend are communicating as I expect
- I will constantly keep an eye on the state of the server to make sure that the data sent there by the frontend is saved as I expect
- I will keep an eye on the database: does the backend save data there in the right format
- I will progress in small steps
- *I will write lots of* `console.log` *statements to make sure I understand how the code and the tests behave and to help pinpoint problems*
- If my code does not work, I will not write more code. Instead, I start deleting the code until it works or just return to a state when everything is still working
- *If a test does not pass, I make sure that the tested functionality for sure works in the application*
- When I ask for help in the course Discord or Telegram channel or elsewhere I formulate my questions properly, see here how to ask for help

Exercises 4.8.-4.12.

Warning: If you find yourself using async/await and *then* methods in the same code, it is almost guaranteed that you are doing something wrong. Use one or the other and don't mix the two.

4.8: Blog List Tests, step 1

Use the SuperTest library for writing a test that makes an HTTP GET request to the */api/blogs* URL. Verify that the blog list application returns the correct amount of blog posts in the JSON format.

Once the test is finished, refactor the route handler to use the async/await syntax instead of promises.

Notice that you will have to make similar changes to the code that were made in the material, like defining the test environment so that you can write tests that use separate databases.

NB: when you are writing your tests ***it is better to not execute them all***, only execute the ones you are working on. Read more about this here.

4.9: Blog List Tests, step 2

Write a test that verifies that the unique identifier property of the blog posts is named *id*, by default the database names the property *_id*.

Make the required changes to the code so that it passes the test. The toJSON method discussed in part 3 is an appropriate place for defining the *id* parameter.

4.10: Blog List Tests, step 3

Write a test that verifies that making an HTTP POST request to the */api/blogs* URL successfully creates a new blog post. At the very least, verify that the total number of blogs in the system is increased by one. You can also verify that the content of the blog post is saved correctly to the database.

Once the test is finished, refactor the operation to use async/await instead of promises.

4.11*: Blog List Tests, step 4

Write a test that verifies that if the *likes* property is missing from the request, it will default to the value 0. Do not test the other properties of the created blogs yet.

Make the required changes to the code so that it passes the test.

4.12*: Blog List tests, step 5

Write tests related to creating new blogs via the */api/blogs* endpoint, that verify that if the *title* or *url* properties are missing from the request data, the backend responds to the request with the status code *400 Bad Request*.

Make the required changes to the code so that it passes the test.

Refactoring tests

Our test coverage is currently lacking. Some requests like *GET /api/notes/:id* and *DELETE /api/notes/:id* aren't tested when the request is sent with an invalid id. The grouping and organization of tests could also use some improvement, as all tests exist on the same "top level" in the test file. The readability of the test would improve if we group related tests with *describe* blocks.

Below is an example of the test file after making some minor improvements:

```
const { test, after, beforeEach, describe } = require('node:test')
const assert = require('node:assert')
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')
const api = supertest(app)

const helper = require('./test_helper')

const Note = require('../models/note')

describe('when there is initially some notes saved', () => {
  beforeEach(async () => {
```

```
  await Note.deleteMany({})
  await Note.insertMany(helper.initialNotes)
})

test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
})

test('all notes are returned', async () => {
  const response = await api.get('/api/notes')

  assert.strictEqual(response.body.length, helper.initialNotes.length)
})

test('a specific note is within the returned notes', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)
  assert(contents.includes('Browser can execute only JavaScript'))
})

describe('viewing a specific note', () => {

  test('succeeds with a valid id', async () => {
    const notesAtStart = await helper.notesInDb()

    const noteToView = notesAtStart[0]

    const resultNote = await api
      .get(`/api/notes/${noteToView.id}`)
      .expect(200)
      .expect('Content-Type', /application\/json/)

    assert.deepStrictEqual(resultNote.body, noteToView)
  })

  test('fails with statuscode 404 if note does not exist', async () => {
    const validNonexistingId = await helper.nonExistingId()

    await api
      .get(`/api/notes/${validNonexistingId}`)
      .expect(404)
  })

  test('fails with statuscode 400 id is invalid', async () => {
    const invalidId = '5a3d5da59070081a82a3445'

    await api
      .get(`/api/notes/${invalidId}`)
      .expect(400)
  })
```

```
})

describe('addition of a new note', () => {
  test('succeeds with valid data', async () => {
    const newNote = {
      content: 'async/await simplifies making async calls',
      important: true,
    }

    await api
      .post('/api/notes')
      .send(newNote)
      .expect(201)
      .expect('Content-Type', /application\/json/)

    const notesAtEnd = await helper.notesInDb()
    assert.strictEqual(notesAtEnd.length, helper.initialNotes.length + 1)

    const contents = notesAtEnd.map(n => n.content)
    assert(contents.includes('async/await simplifies making async calls'))
  })

  test('fails with status code 400 if data invalid', async () => {
    const newNote = {
      important: true
    }

    await api
      .post('/api/notes')
      .send(newNote)
      .expect(400)

    const notesAtEnd = await helper.notesInDb()

    assert.strictEqual(notesAtEnd.length, helper.initialNotes.length)
  })
})

describe('deletion of a note', () => {
  test('succeeds with status code 204 if id is valid', async () => {
    const notesAtStart = await helper.notesInDb()
    const noteToDelete = notesAtStart[0]

    await api
      .delete(`/api/notes/${noteToDelete.id}`)
      .expect(204)

    const notesAtEnd = await helper.notesInDb()

    assert.strictEqual(notesAtEnd.length, helper.initialNotes.length - 1)

    const contents = notesAtEnd.map(r => r.content)
    assert(!contents.includes(noteToDelete.content))
  })
```

```
  })
})

after(async () => {
  await mongoose.connection.close()
})
```

The test output in the console is grouped according to the *describe* blocks:

```
▶ when there is initially some notes saved
  ✔ notes are returned as json (784.923458ms)
  ✔ all notes are returned (61.210958ms)
  ✔ a specific note is within the returned notes (54.483334ms)
  ▶ viewing a specific note
    ✔ succeeds with a valid id (75.398542ms)
    ✔ fails with statuscode 404 if note does not exist (128.202209ms)
    ✔ fails with statuscode 400 id is invalid (56.257166ms)
  ▶ viewing a specific note (260.737375ms)

  ▶ addition of a new note
    ✔ succeeds with valid data (103.98325ms)
    ✔ fails with status code 400 if data invalid (62.044916ms)
  ▶ addition of a new note (166.798125ms)

  ▶ deletion of a note
    ✔ succeeds with status code 204 if id is valid (152.118417ms)
  ▶ deletion of a note (152.820417ms)

▶ when there is initially some notes saved (1482.8545ms)
```

There is still room for improvement, but it is time to move forward.

This way of testing the API, by making HTTP requests and inspecting the database with Mongoose, is by no means the only nor the best way of conducting API-level integration tests for server applications. There is no universal best way of writing tests, as it all depends on the application being tested and available resources.

You can find the code for our current application in its entirety in the *part4-6* branch of this GitHub repository.

Exercises 4.13.-4.14.

4.13 Blog List Expansions, step 1

Implement functionality for deleting a single blog post resource.

Use the async/await syntax. Follow RESTful conventions when defining the HTTP API.

Implement tests for the functionality.

4.14 Blog List Expansions, step 2

Implement functionality for updating the information of an individual blog post.

Use async/await.

The application mostly needs to update the number of *likes* for a blog post. You can implement this functionality the same way that we implemented updating notes in part 3 .

Implement tests for the functionality.

Propose changes to material

Part 4a
Previous part

Part 4c
Next part

About course

Course contents

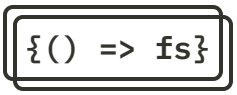
FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI

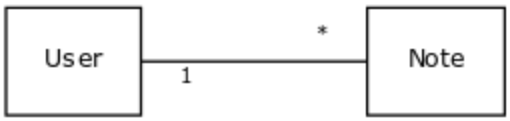

HOUSTON



c User administration

We want to add user authentication and authorization to our application. Users should be stored in the database and every note should be linked to the user who created it. Deleting and editing a note should only be allowed for the user who created it.

Let's start by adding information about users to the database. There is a one-to-many relationship between the user (*User*) and notes (*Note*):



If we were working with a relational database the implementation would be straightforward. Both resources would have their separate database tables, and the id of the user who created a note would be stored in the notes table as a foreign key.

When working with document databases the situation is a bit different, as there are many different ways of modeling the situation.

The existing solution saves every note in the *notes collection* in the database. If we do not want to change this existing collection, then the natural choice is to save users in their own collection, *users* for example.

Like with all document databases, we can use object IDs in Mongo to reference documents in other collections. This is similar to using foreign keys in relational databases.

Traditionally document databases like Mongo do not support *join queries* that are available in relational databases, used for aggregating data from multiple tables. However, starting from version 3.2. Mongo

has supported lookup aggregation queries. We will not be taking a look at this functionality in this course.

If we need functionality similar to join queries, we will implement it in our application code by making multiple queries. In certain situations, Mongoose can take care of joining and aggregating data, which gives the appearance of a join query. However, even in these situations, Mongoose makes multiple queries to the database in the background.

References across collections

If we were using a relational database the note would contain a *reference key* to the user who created it. In document databases, we can do the same thing.

Let's assume that the *users* collection contains two users:

```
[
  {
    username: 'mluukkai',
    _id: 123456,
  },
  {
    username: 'hellas',
    _id: 141414,
  },
]
```

The *notes* collection contains three notes that all have a *user* field that references a user in the *users* collection:

```
[
  {
    content: 'HTML is easy',
    important: false,
    _id: 221212,
    user: 123456,
  },
  {
    content: 'The most important operations of HTTP protocol are GET and POST',
    important: true,
    _id: 221255,
    user: 123456,
  },
  {
    content: 'A proper dinosaur codes with Java',
    important: false,
    _id: 221244,
    user: 141414,
```

```
  },
]
```

Document databases do not demand the foreign key to be stored in the note resources, it could *also* be stored in the users collection, or even both:

```
[
  {
    username: 'mluukkai',
    _id: 123456,
    notes: [221212, 221255],
  },
  {
    username: 'hellas',
    _id: 141414,
    notes: [221244],
  },
]
```

Since users can have many notes, the related ids are stored in an array in the *notes* field.

Document databases also offer a radically different way of organizing the data: In some situations, it might be beneficial to nest the entire notes array as a part of the documents in the users collection:

```
[
  {
    username: 'mluukkai',
    _id: 123456,
    notes: [
      {
        content: 'HTML is easy',
        important: false,
      },
      {
        content: 'The most important operations of HTTP protocol are GET and POST',
        important: true,
      },
    ],
  },
  {
    username: 'hellas',
    _id: 141414,
    notes: [
      {
        content:
          'A proper dinosaur codes with Java',
        important: false,
      },
    ],
```

```
  },
]
```

In this schema, notes would be tightly nested under users and the database would not generate ids for them.

The structure and schema of the database are not as self-evident as it was with relational databases. The chosen schema must support the use cases of the application the best. This is not a simple design decision to make, as all use cases of the applications are not known when the design decision is made.

Paradoxically, schema-less databases like Mongo require developers to make far more radical design decisions about data organization at the beginning of the project than relational databases with schemas. On average, relational databases offer a more or less suitable way of organizing data for many applications.

Mongoose schema for users

In this case, we decide to store the ids of the notes created by the user in the user document. Let's define the model for representing a user in the *models/user.js* file:

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

const userSchema = new mongoose.Schema({
  username: String,
  name: String,
  passwordHash: String,
  notes: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Note'
    }
  ],
})

userSchema.set('toJSON', {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString()
    delete returnedObject._id
    delete returnedObject.__v
    // the passwordHash should not be revealed
    delete returnedObject.passwordHash
  }
})

const User = mongoose.model('User', userSchema)

module.exports = User
```

The ids of the notes are stored within the user document as an array of Mongo ids. The definition is as follows:

```
{
  type: mongoose.Schema.Types.ObjectId,
  ref: 'Note'
}
```

The type of the field is *ObjectId* that references *note*-style documents. Mongo does not inherently know that this is a field that references notes, the syntax is purely related to and defined by Mongoose.

Let's expand the schema of the note defined in the *models/note.js* file so that the note contains information about the user who created it:

```
const noteSchema = new mongoose.Schema({
  content: {
    type: String,
    required: true,
    minlength: 5
  },
  important: Boolean,
  user: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  }
})
```

In stark contrast to the conventions of relational databases, *references are now stored in both documents*: the note references the user who created it, and the user has an array of references to all of the notes created by them.

Creating users

Let's implement a route for creating new users. Users have a unique *username*, a *name* and something called a *passwordHash*. The password hash is the output of a one-way hash function applied to the user's password. It is never wise to store unencrypted plain text passwords in the database!

Let's install the bcrypt package for generating the password hashes:

```
npm install bcrypt
```

Creating new users happens in compliance with the RESTful conventions discussed in part 3, by making an HTTP POST request to the *users* path.

Let's define a separate *router* for dealing with users in a new *controllers/users.js* file. Let's take the router into use in our application in the *app.js* file, so that it handles requests made to the */api/users* url:

```
const usersRouter = require('./controllers/users')

// ...

app.use('/api/users', usersRouter)
```

The contents of the file, *controllers/users.js*, that defines the router is as follows:

```
const bcrypt = require('bcrypt')
const usersRouter = require('express').Router()
const User = require('../models/user')

usersRouter.post('/', async (request, response) => {
  const { username, name, password } = request.body

  const saltRounds = 10
  const passwordHash = await bcrypt.hash(password, saltRounds)

  const user = new User({
    username,
    name,
    passwordHash,
  })

  const savedUser = await user.save()

  response.status(201).json(savedUser)
})

module.exports = usersRouter
```

The password sent in the request is *not* stored in the database. We store the *hash* of the password that is generated with the `bcrypt.hash` function.

The fundamentals of storing passwords are outside the scope of this course material. We will not discuss what the magic number 10 assigned to the saltRounds variable means, but you can read more about it in the linked material.

Our current code does not contain any error handling or input validation for verifying that the username and password are in the desired format.

The new feature can and should initially be tested manually with a tool like Postman. However testing things manually will quickly become too cumbersome, especially once we implement functionality that enforces usernames to be unique.

It takes much less effort to write automated tests, and it will make the development of our application much easier.

Our initial tests could look like this:

```
const bcrypt = require('bcrypt')
const User = require('../models/user')

//...

describe('when there is initially one user in db', () => {
  beforeEach(async () => {
    await User.deleteMany({})

    const passwordHash = await bcrypt.hash('sekret', 10)
    const user = new User({ username: 'root', passwordHash })

    await user.save()
  })

  test('creation succeeds with a fresh username', async () => {
    const usersAtStart = await helper.usersInDb()

    const newUser = {
      username: 'mluukkai',
      name: 'Matti Luukkainen',
      password: 'salainen',
    }

    await api
      .post('/api/users')
      .send(newUser)
      .expect(201)
      .expect('Content-Type', /application\/json/)

    const usersAtEnd = await helper.usersInDb()
    assert.strictEqual(usersAtEnd.length, usersAtStart.length + 1)

    const usernames = usersAtEnd.map(u => u.username)
    assert(usernames.includes(newUser.username))
  })
})
```

The tests use the *usersInDb()* helper function that we implemented in the *tests/test_helper.js* file. The function is used to help us verify the state of the database after a user is created:

```
const User = require('../models/user')

// ...
```

```
const usersInDb = async () => {
  const users = await User.find({})
  return users.map(u => u.toJSON())
}

module.exports = {
  initialNotes,
  nonExistingId,
  notesInDb,
  usersInDb,
}
```

The *beforeEach* block adds a user with the username *root* to the database. We can write a new test that verifies that a new user with the same username can not be created:

```
describe('when there is initially one user in db', () => {
  // ...

  test('creation fails with proper statuscode and message if username already taken',
async () => {
    const usersAtStart = await helper.usersInDb()

    const newUser = {
      username: 'root',
      name: 'Superuser',
      password: 'salainen',
    }

    const result = await api
      .post('/api/users')
      .send(newUser)
      .expect(400)
      .expect('Content-Type', /application\/json/)

    const usersAtEnd = await helper.usersInDb()
    assert(result.body.error.includes('expected `username` to be unique'))

    assert.strictEqual(usersAtEnd.length, usersAtStart.length)
  })
})
```

The test case obviously will not pass at this point. We are essentially practicing test-driven development (TDD), where tests for new functionality are written before the functionality is implemented.

Mongoose validations do not provide a direct way to check the uniqueness of a field value. However, it is possible to achieve uniqueness by defining uniqueness index for a field. The definition is done as follows:

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

```
const userSchema = mongoose.Schema({
  username: {
    type: String,
    required: true,
    unique: true // this ensures the uniqueness of username
  },
  name: String,
  passwordHash: String,
  notes: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Note'
    }
  ],
})

// ...
```

However, we want to be careful when using the uniqueness index. If there are already documents in the database that violate the uniqueness condition, no index will be created. So when adding a uniqueness index, make sure that the database is in a healthy state! The test above added the user with username `root` to the database twice, and these must be removed for the index to be formed and the code to work.

Mongoose validations do not detect the index violation, and instead of `ValidationError` they return an error of type `MongoServerError`. We therefore need to extend the error handler for that case:

```
const errorHandler = (error, request, response, next) => {
  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 })
  } else if (error.name === 'MongoServerError' && error.message.includes('E11000
duplicate key error')) {
    return response.status(400).json({ error: 'expected `username` to be unique' })
  }

  next(error)
}
```

After these changes, the tests will pass.

We could also implement other validations into the user creation. We could check that the username is long enough, that the username only consists of permitted characters, or that the password is strong enough. Implementing these functionalities is left as an optional exercise.

Before we move onward, let's add an initial implementation of a route handler that returns all of the users in the database:

```
usersRouter.get('/', async (request, response) => {
  const users = await User.find({})
  response.json(users)
})
```

copy

For making new users in a production or development environment, you may send a POST request to `/api/users/` via Postman or REST Client in the following format:

```
{
    "username": "root",
    "name": "Superuser",
    "password": "salainen"
}
```

copy

The list looks like this:

You can find the code for our current application in its entirety in the *part4-7* branch of this GitHub repository.

Creating a new note

The code for creating a new note has to be updated so that the note is assigned to the user who created it.

Let's expand our current implementation in *controllers/notes.js* so that the information about the user who created a note is sent in the *userId* field of the request body:

```
const User = require('../models/user')
```

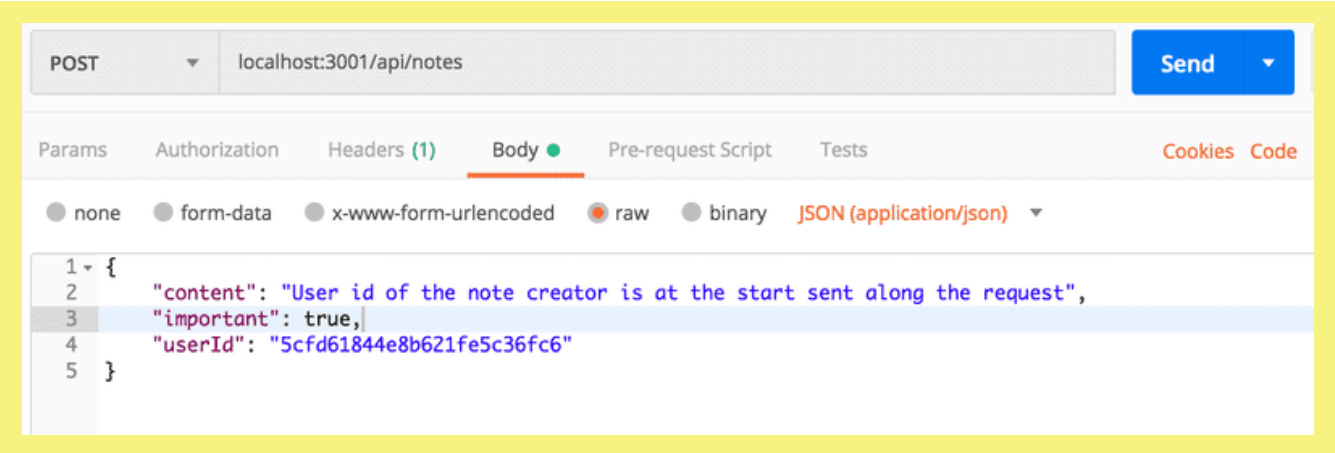
copy

```
//...

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

  const user = await User.findById(body.userId)

  const note = new Note({
    content: body.content,
    important: body.important === undefined ? false : body.important,
    user: user.id
  })

  const savedNote = await note.save()
  user.notes = user.notes.concat(savedNote._id)
  await user.save()

  response.status(201).json(savedNote)
})
```

It's worth noting that the *user* object also changes. The *id* of the note is stored in the *notes* field of the *user* object:

```
const user = await User.findById(body.userId)

// ...

user.notes = user.notes.concat(savedNote._id)
await user.save()
```

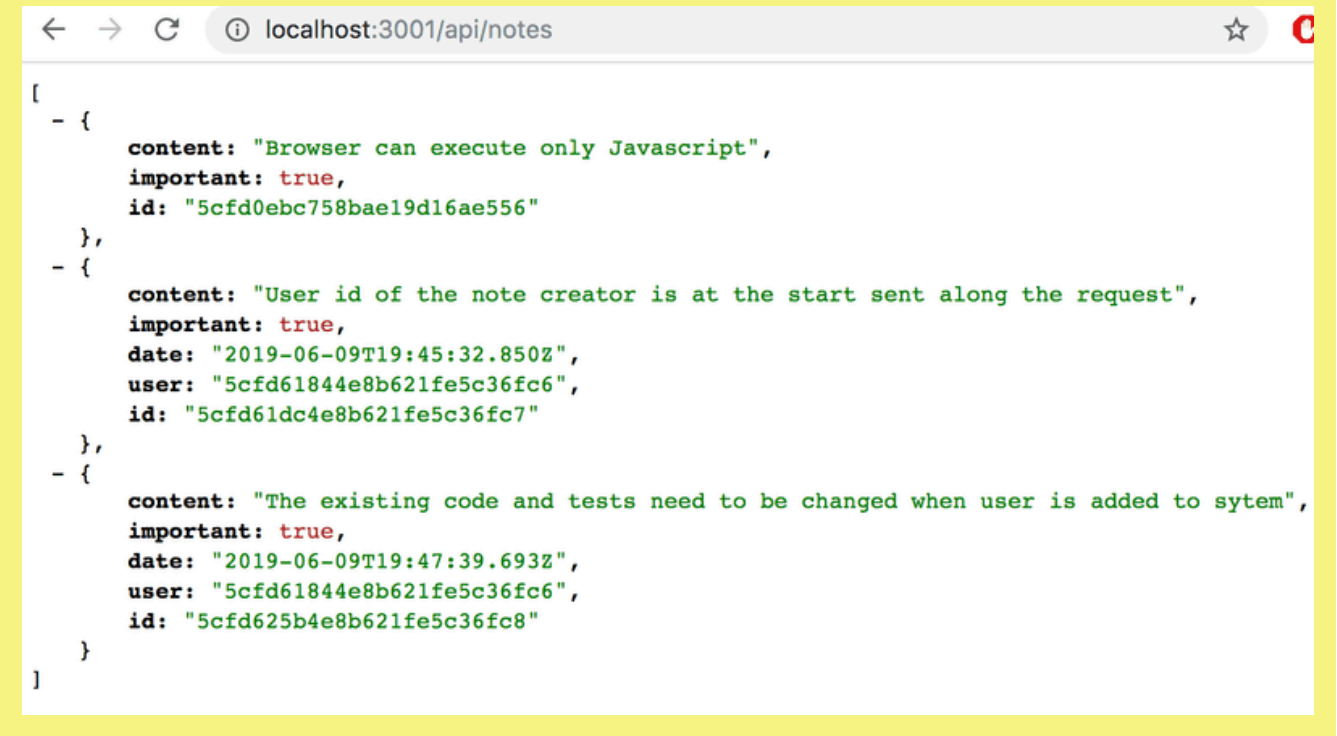
Let's try to create a new note



The operation appears to work. Let's add one more note and then visit the route for fetching all users:

We can see that the user has two notes.

Likewise, the ids of the users who created the notes can be seen when we visit the route for fetching all notes:



Populate

We would like our API to work in such a way, that when an HTTP GET request is made to the */api/users* route, the user objects would also contain the contents of the user's notes and not just their id. In a relational database, this functionality would be implemented with a *join query*.

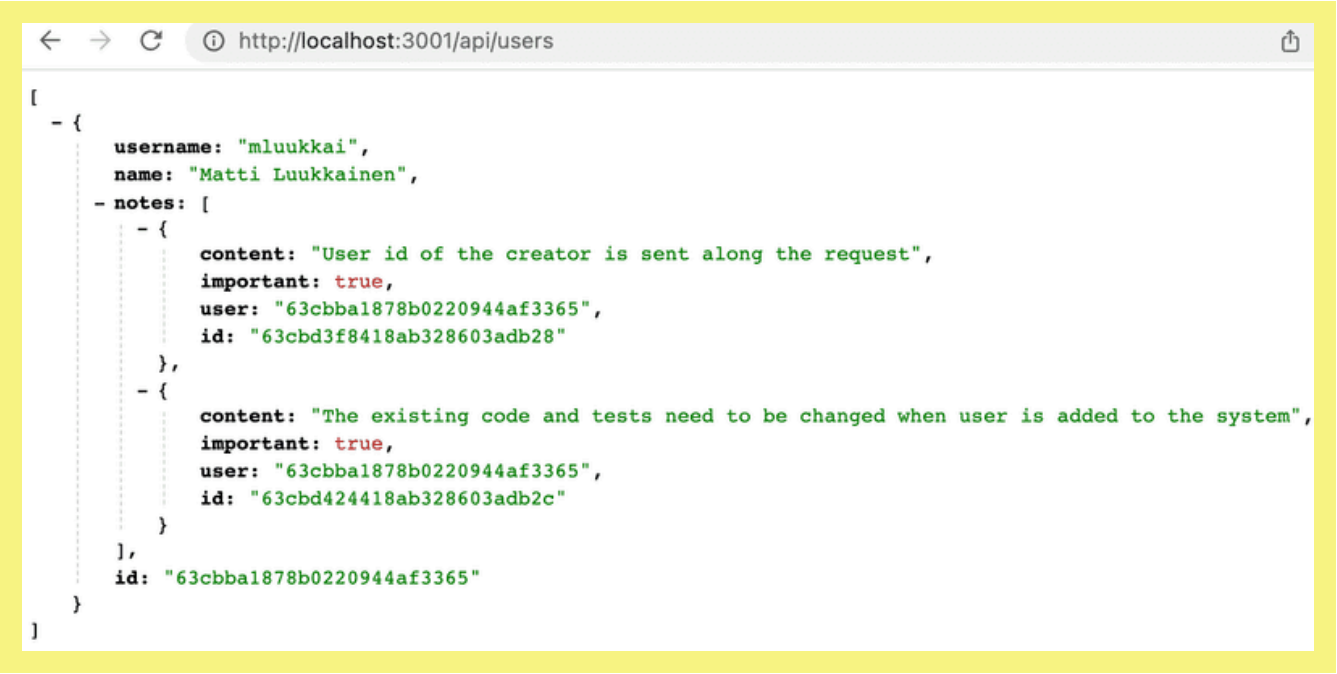
As previously mentioned, document databases do not properly support join queries between collections, but the Mongoose library can do some of these joins for us. Mongoose accomplishes the join by doing multiple queries, which is different from join queries in relational databases which are *transactional*, meaning that the state of the database does not change during the time that the query is made. With join queries in Mongoose, nothing can guarantee that the state between the collections being joined is consistent, meaning that if we make a query that joins the user and notes collections, the state of the collections may change during the query.

The Mongoose join is done with the `populate` method. Let's update the route that returns all users first in *controllers/users.js* file:

```
usersRouter.get('/', async (request, response) => {
  const users = await User
    .find({}).populate('notes')

  response.json(users)
})
```

The `populate` method is chained after the *find* method making the initial query. The argument given to the populate method defines that the *ids* referencing *note* objects in the *notes* field of the *user* document will be replaced by the referenced *note* documents.

The result is almost exactly what we wanted:



We can use the populate method for choosing the fields we want to include from the documents. In addition to the field *id* we are now only interested in *content* and *important*.

The selection of fields is done with the Mongo `syntax`:

```
usersRouter.get('/', async (request, response) => {
  const users = await User
    .find({}).populate('notes', { content: 1, important: 1 })

  response.json(users)
})
```

The result is now exactly like we want it to be:

Let's also add a suitable population of user information to notes in the *controllers/notes.js* file:

```
notesRouter.get('/', async (request, response) => {
  const notes = await Note
    .find({}).populate('user', { username: 1, name: 1 })

  response.json(notes)
})
```

Now the user's information is added to the *user* field of note objects.

It's important to understand that the database does not know that the ids stored in the *user* field of the notes collection reference documents in the user collection.

The functionality of the *populate* method of Mongoose is based on the fact that we have defined "types" to the references in the Mongoose schema with the *ref* option:

```
const noteSchema = new mongoose.Schema({
  content: {
    type: String,
    required: true,
    minlength: 5
  },
  important: Boolean,
  user: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  }
})
```

You can find the code for our current application in its entirety in the *part4-8* branch of this GitHub repository.

NOTE: At this stage, firstly, some tests will fail. We will leave fixing the tests to a non-compulsory exercise. Secondly, in the deployed notes app, the creating a note feature will stop working as user is not yet linked to the frontend.

Propose changes to material

Part 4b Part 4d

Previous part Next part

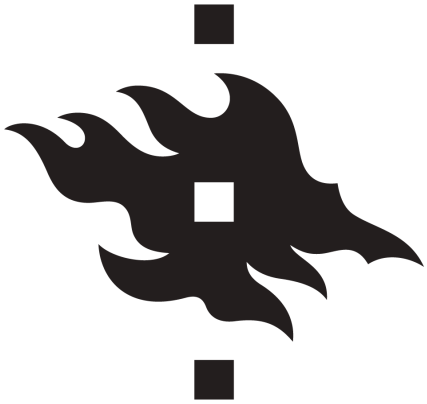
About course

Course contents

FAQ

Partners

Challenge



UNIVERSITY OF HELSINKI


HOUSTON

{() => fs}

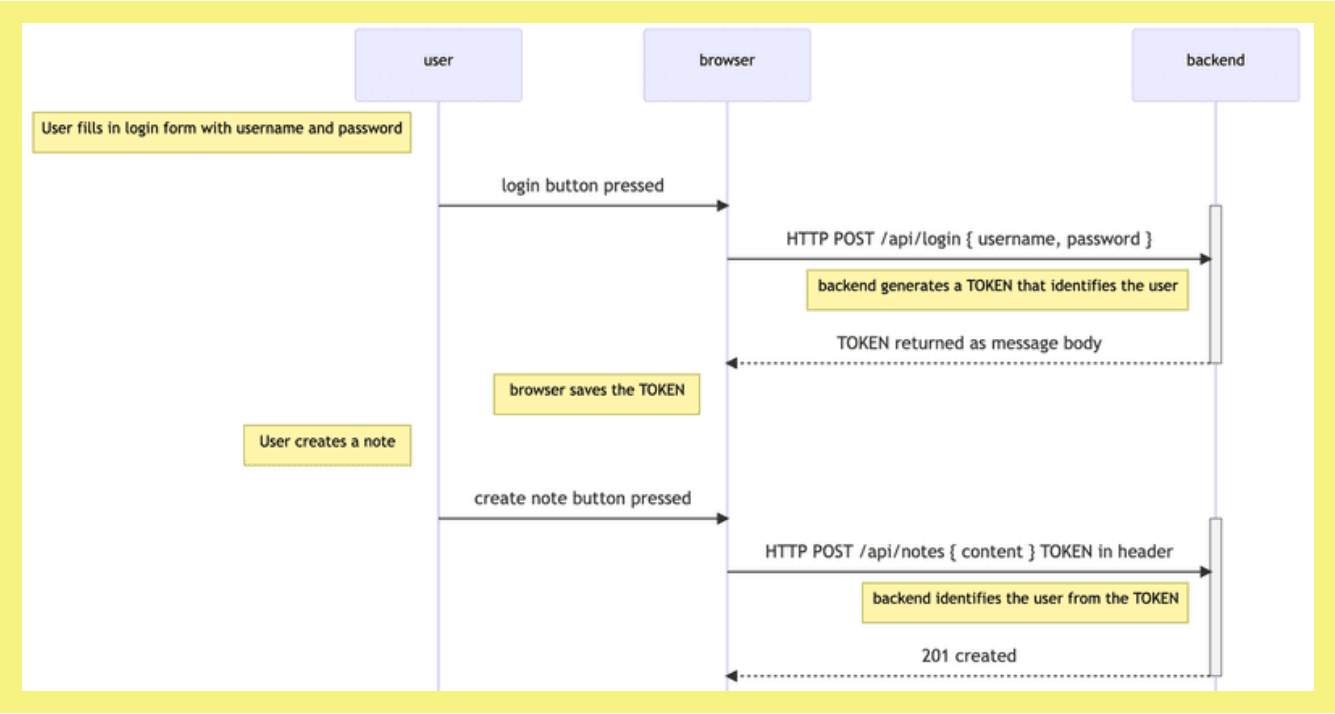
Fullstack > Part 4 > Token authentication

d Token authentication

Users must be able to log into our application, and when a user is logged in, their user information must automatically be attached to any new notes they create.

We will now implement support for token-based authentication to the backend.

The principles of token-based authentication are depicted in the following sequence diagram:



- User starts by logging in using a login form implemented with React
  - We will add the login form to the frontend in part 5

- This causes the React code to send the username and the password to the server address */api/login* as an HTTP POST request.
- If the username and the password are correct, the server generates a *token* that somehow identifies the logged-in user.
  - The token is signed digitally, making it impossible to falsify (with cryptographic means)
- The backend responds with a status code indicating the operation was successful and returns the token with the response.
- The browser saves the token, for example to the state of a React application.
- When the user creates a new note (or does some other operation requiring identification), the React code sends the token to the server with the request.
- The server uses the token to identify the user

Let's first implement the functionality for logging in. Install the `jsonwebtoken` library, which allows us to generate JSON web tokens.

```
npm install jsonwebtoken
```

copy

The code for login functionality goes to the file *controllers/login.js*.

```
const jwt = require('jsonwebtoken')
const bcrypt = require('bcrypt')
const loginRouter = require('express').Router()
const User = require('../models/user')

loginRouter.post('/', async (request, response) => {
  const { username, password } = request.body

  const user = await User.findOne({ username })
  const passwordCorrect = user === null
    ? false
    : await bcrypt.compare(password, user.passwordHash)

  if (!(user && passwordCorrect)) {
    return response.status(401).json({
      error: 'invalid username or password'
    })
  }

  const userForToken = {
    username: user.username,
    id: user._id,
  }

  const token = jwt.sign(userForToken, process.env.SECRET)
```

copy

```
  response
    .status(200)
    .send({ token, username: user.username, name: user.name })
})

module.exports = loginRouter
```

The code starts by searching for the user from the database by the *username* attached to the request.

```
const user = await User.findOne({ username })
```

Next, it checks the *password*, also attached to the request.

```
const passwordCorrect = user === null
  ? false
  : await bcrypt.compare(password, user.passwordHash)
```

Because the passwords themselves are not saved to the database, but *hashes* calculated from the passwords, the `bcrypt.compare` method is used to check if the password is correct:

```
await bcrypt.compare(password, user.passwordHash)
```

If the user is not found, or the password is incorrect, the request is responded with the status code 401 unauthorized. The reason for the failure is explained in the response body.

```
if (!(user && passwordCorrect)) {
  return response.status(401).json({
    error: 'invalid username or password'
  })
}
```

If the password is correct, a token is created with the method `jwt.sign`. The token contains the username and the user id in a digitally signed form.

```
const userForToken = {
  username: user.username,
  id: user._id,
}
```

```
const token = jwt.sign(userForToken, process.env.SECRET)
```

The token has been digitally signed using a string from the environment variable *SECRET* as the *secret*. The digital signature ensures that only parties who know the secret can generate a valid token. The value for the environment variable must be set in the *.env* file.

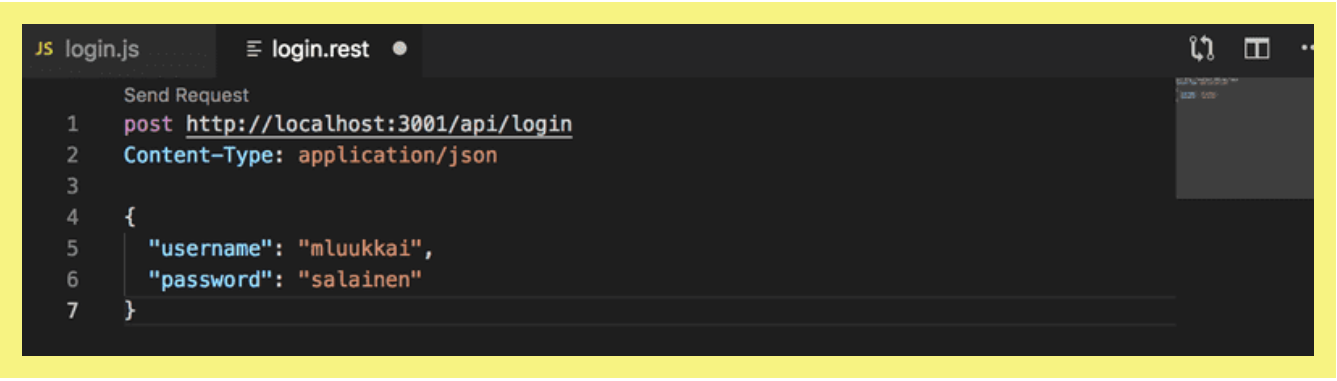
A successful request is responded to with the status code *200 OK*. The generated token and the username of the user are sent back in the response body.

```
response
  .status(200)
  .send({ token, username: user.username, name: user.name })
```

Now the code for login just has to be added to the application by adding the new router to *app.js*.

```
const loginRouter = require('./controllers/login')

//...

app.use('/api/login', loginRouter)
```

Let's try logging in using VS Code REST-client:



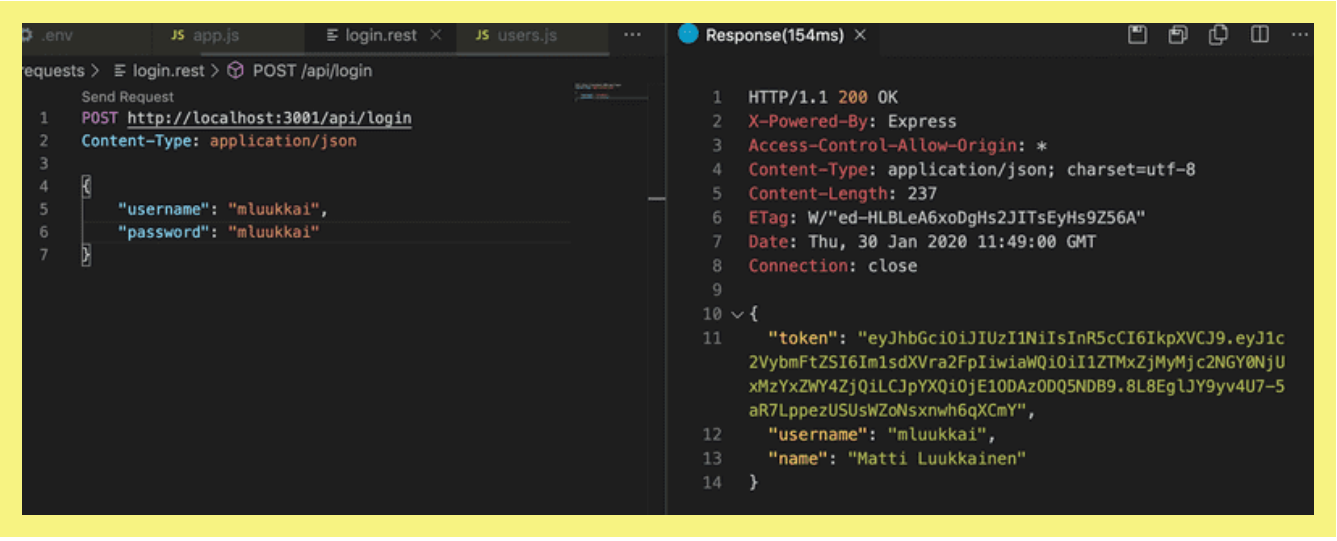
It does not work. The following is printed to the console:

```
(node:32911) UnhandledPromiseRejectionWarning: Error: secretOrPrivateKey must have a
value
    at Object.module.exports [as sign] (/Users/mluukkai/opetus/_2019fullstack-
koodit/osa3/notes-backend/node_modules/jsonwebtoken/sign.js:101:20)
    at loginRouter.post (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-
backend/controllers/login.js:26:21)
(node:32911) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error
```

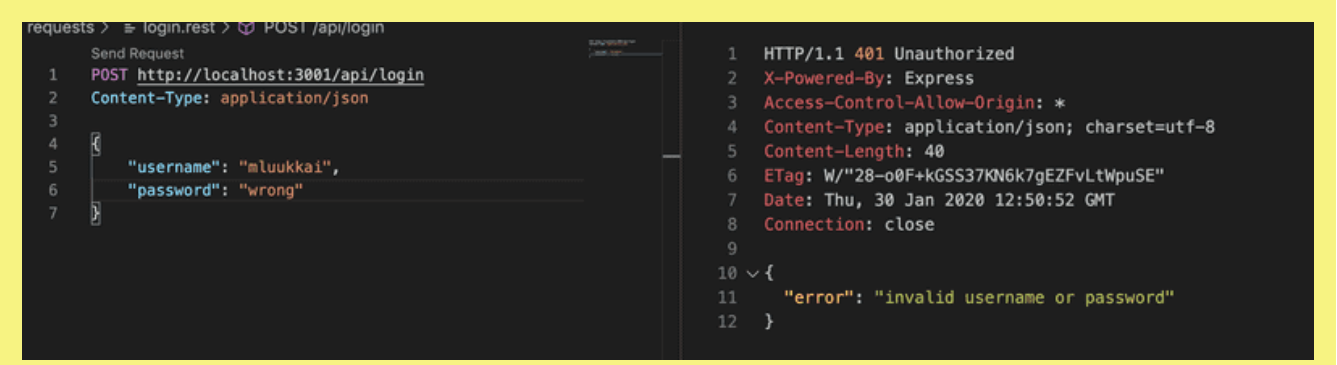
```
originated either by throwing inside of an async function without a catch block, or by
rejecting a promise which was not handled with .catch(). (rejection id: 2)
```

The command `jwt.sign(userForToken, process.env.SECRET)` fails. We forgot to set a value to the environment variable *SECRET*. It can be any string. When we set the value in file *.env* (and restart the server), the login works.

A successful login returns the user details and the token:



A wrong username or password returns an error message and the proper status code:



Limiting creating new notes to logged-in users

Let's change creating new notes so that it is only possible if the post request has a valid token attached. The note is then saved to the notes list of the user identified by the token.

There are several ways of sending the token from the browser to the server. We will use the `Authorization` header. The header also tells which authentication scheme is used. This can be necessary if the server offers multiple ways to authenticate. Identifying the scheme tells the server how the attached credentials should be interpreted.

The *Bearer* scheme is suitable for our needs.

In practice, this means that if the token is, for example, the string *eyJhbGciOiJIUzI1NiIsInR5c2VybmFtZSI6Im1sdXVra2FpIiwiaW*, the Authorization header will have the value:

```
Bearer eyJhbGciOiJIUzI1NiIsInR5c2VybmFtZSI6Im1sdXVra2FpIiwiaW
```

copy

Creating new notes will change like so (*controllers/notes.js*):

```
const jwt = require('jsonwebtoken')

// ...
const getTokenFrom = request => {
  const authorization = request.get('authorization')
  if (authorization && authorization.startsWith('Bearer ')) {
    return authorization.replace('Bearer ', '')
  }
  return null
}

notesRouter.post('/', async (request, response) => {
  const body = request.body
  const decodedToken = jwt.verify(getTokenFrom(request), process.env.SECRET)
  if (!decodedToken.id) {
    return response.status(401).json({ error: 'token invalid' })
  }
  const user = await User.findById(decodedToken.id)

  const note = new Note({
    content: body.content,
    important: body.important === undefined ? false : body.important,
    user: user._id
  })

  const savedNote = await note.save()
  user.notes = user.notes.concat(savedNote._id)
  await user.save()

  response.json(savedNote)
})
```

copy

The helper function `getTokenFrom` isolates the token from the *authorization* header. The validity of the token is checked with `jwt.verify`. The method also decodes the token, or returns the Object which the token was based on.

```
const decodedToken = jwt.verify(token, process.env.SECRET)
```

copy

If the token is missing or it is invalid, the exception *JsonWebTokenError* is raised. We need to extend the error handling middleware to take care of this particular case:

```
const errorHandler = (error, request, response, next) => {
  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 })
  } else if (error.name === 'MongoServerError' && error.message.includes('E11000
duplicate key error')) {
    return response.status(400).json({ error: 'expected `username` to be unique' })
  } else if (error.name ===  'JsonWebTokenError') {
    return response.status(401).json({ error: 'token invalid' })
  }

  next(error)
}
```

The object decoded from the token contains the *username* and *id* fields, which tell the server who made the request.

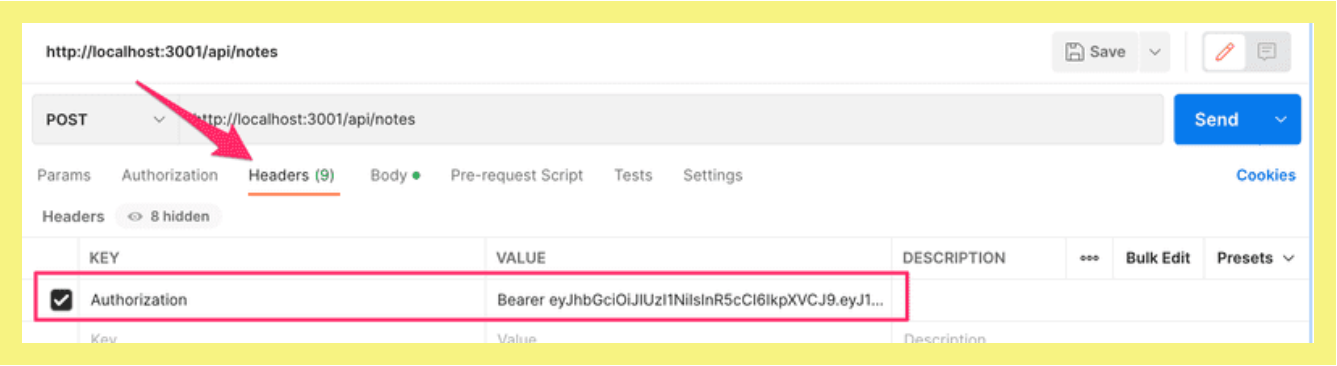
If the object decoded from the token does not contain the user's identity (`decodedToken.id` is undefined), error status code 401 unauthorized is returned and the reason for the failure is explained in the response body.

```
if (!decodedToken.id) {
  return response.status(401).json({
    error: 'token invalid'
  })
}
```

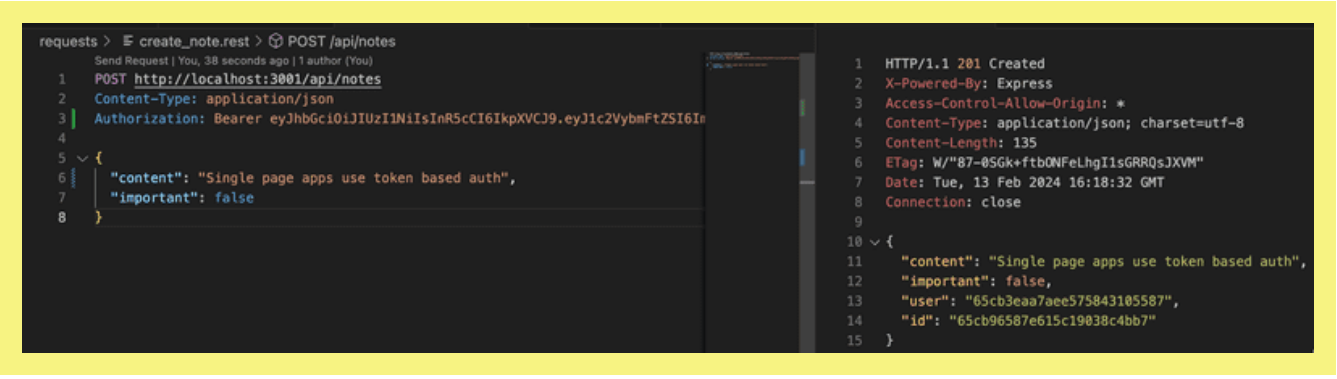
When the identity of the maker of the request is resolved, the execution continues as before.

A new note can now be created using Postman if the *authorization* header is given the correct value, the string *Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ*, where the second value is the token returned by the *login* operation.

Using Postman this looks as follows:



and with Visual Studio Code REST client



Current application code can be found on GitHub, branch *part4-9*.

If the application has multiple interfaces requiring identification, JWT's validation should be separated into its own middleware. An existing library like express-jwt could also be used.

Problems of Token-based authentication

Token authentication is pretty easy to implement, but it contains one problem. Once the API user, eg. a React app gets a token, the API has a blind trust to the token holder. What if the access rights of the token holder should be revoked?

There are two solutions to the problem. The easier one is to limit the validity period of a token:

```
loginRouter.post('/', async (request, response) => {
  const { username, password } = request.body

  const user = await User.findOne({ username })
  const passwordCorrect = user === null
    ? false
    : await bcrypt.compare(password, user.passwordHash)

  if (!(user && passwordCorrect)) {
    return response.status(401).json({
      error: 'invalid username or password'
    })
  }
```

```
  const userForToken = {
    username: user.username,
    id: user._id,
  }

  // token expires in 60*60 seconds, that is, in one hour
  const token = jwt.sign(
    userForToken,
    process.env.SECRET,
    { expiresIn: 60*60 }
  )

  response
    .status(200)
    .send({ token, username: user.username, name: user.name })
})
```

Once the token expires, the client app needs to get a new token. Usually, this happens by forcing the user to re-login to the app.

The error handling middleware should be extended to give a proper error in the case of an expired token:

```
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 })
  } else if (error.name === 'MongoServerError' && error.message.includes('E11000
duplicate key error')) {
    return response.status(400).json({
      error: 'expected `username` to be unique'
    })
  } else if (error.name === 'JsonWebTokenError') {
    return response.status(401).json({
      error: 'invalid token'
    })
  } else if (error.name === 'TokenExpiredError') {
    return response.status(401).json({
      error: 'token expired'
    })
  }

  next(error)
}
```

The shorter the expiration time, the more safe the solution is. So if the token gets into the wrong hands or user access to the system needs to be revoked, the token is only usable for a limited amount of time.

On the other hand, a short expiration time forces a potential pain to a user, one must login to the system more frequently.

The other solution is to save info about each token to the backend database and to check for each API request if the access rights corresponding to the tokens are still valid. With this scheme, access rights can be revoked at any time. This kind of solution is often called a *server-side session*.

The negative aspect of server-side sessions is the increased complexity in the backend and also the effect on performance since the token validity needs to be checked for each API request to the database. Database access is considerably slower compared to checking the validity of the token itself. That is why it is quite common to save the session corresponding to a token to a *key-value database* such as Redis, that is limited in functionality compared to eg. MongoDB or a relational database, but extremely fast in some usage scenarios.

When server-side sessions are used, the token is quite often just a random string, that does not include any information about the user as it is quite often the case when jwt-tokens are used. For each API request, the server fetches the relevant information about the identity of the user from the database. It is also quite usual that instead of using Authorization-header, *cookies* are used as the mechanism for transferring the token between the client and the server.

End notes

There have been many changes to the code which have caused a typical problem for a fast-paced software project: most of the tests have broken. Because this part of the course is already jammed with new information, we will leave fixing the tests to a non-compulsory exercise.

Usernames, passwords and applications using token authentication must always be used over HTTPS. We could use a Node HTTPS server in our application instead of the HTTP server (it requires more configuration). On the other hand, the production version of our application is in Fly.io, so our application stays secure: Fly.io routes all traffic between a browser and the Fly.io server over HTTPS.

We will implement login to the frontend in the next part.

NOTE: At this stage, in the deployed notes app, it is expected that the creating a note feature will stop working as the backend login feature is not yet linked to the frontend.

Exercises 4.15.-4.23.

In the next exercises, the basics of user management will be implemented for the Bloglist application. The safest way is to follow the course material from part 4 chapter User administration to the chapter Token authentication. You can of course also use your creativity.

One more warning: If you notice you are mixing async/await and `then` calls, it is 99% certain you are doing something wrong. Use either or, never both.

4.15: Blog List Expansion, step 3

Implement a way to create new users by doing an HTTP POST request to address *api/users*. Users have a *username, password and name*.

Do not save passwords to the database as clear text, but use the *bcrypt* library like we did in part 4 chapter Creating users.

NB Some Windows users have had problems with *bcrypt*. If you run into problems, remove the library with command

```
npm uninstall bcrypt
```

copy

and install bcryptjs instead.

Implement a way to see the details of all users by doing a suitable HTTP request.

The list of users can, for example, look as follows:

4.16*: Blog List Expansion, step 4

Add a feature which adds the following restrictions to creating new users: Both username and password must be given and both must be at least 3 characters long. The username must be unique.

The operation must respond with a suitable status code and some kind of an error message if an invalid user is created.

NB Do not test password restrictions with Mongoose validations. It is not a good idea because the password received by the backend and the password hash saved to the database are not the same thing. The password length should be validated in the controller as we did in part 3 before using Mongoose validation.

Also, **implement tests** that ensure invalid users are not created and that an invalid add user operation returns a suitable status code and error message.

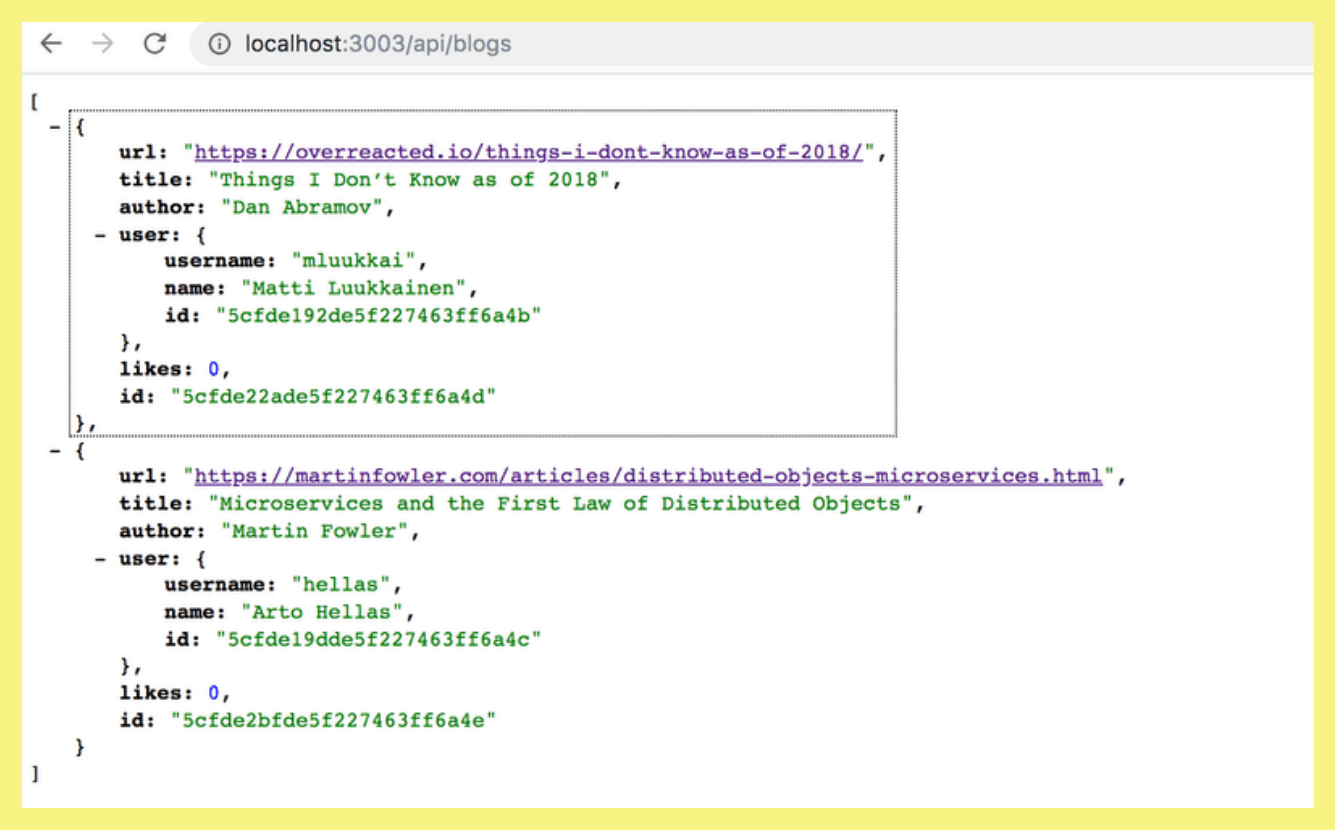
NB if you decide to define tests on multiple files, you should note that by default each test file is executed in its own process (see `Test execution model` in the documentation). The consequence of this is that different test files are executed at the same time. Since the tests share the same database, simultaneous execution may cause problems, which can be avoided by executing the tests with the option `--test-concurrency=1` , i.e. defining them to be executed sequentially.

4.17: Blog List Expansion, step 5

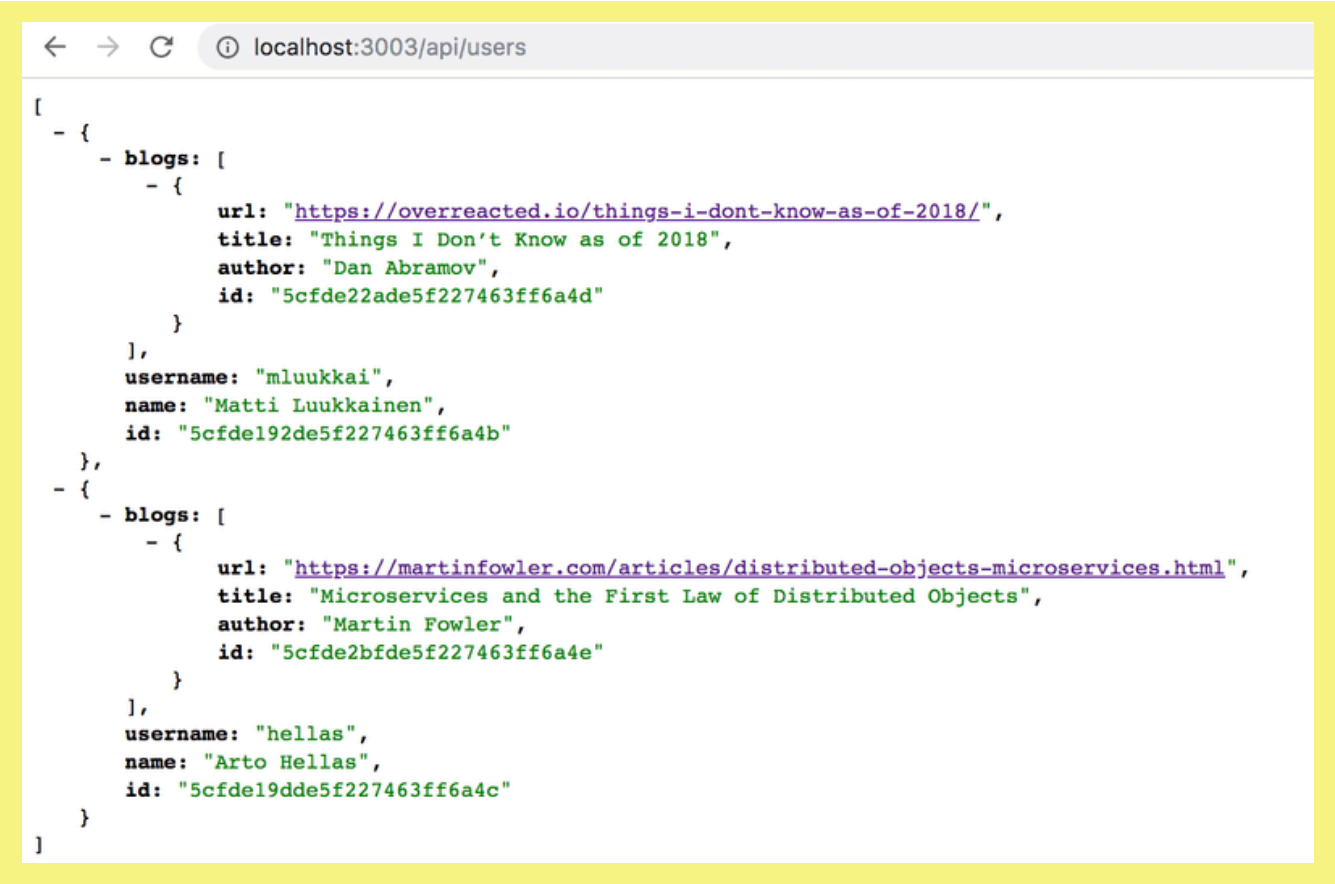
Expand blogs so that each blog contains information on the creator of the blog.

Modify adding new blogs so that when a new blog is created, *any* user from the database is designated as its creator (for example the one found first). Implement this according to part 4 chapter populate. Which user is designated as the creator does not matter just yet. The functionality is finished in exercise 4.19.

Modify listing all blogs so that the creator's user information is displayed with the blog:



and listing all users also displays the blogs created by each user:



4.18: Blog List Expansion, step 6

Implement token-based authentication according to part 4 chapter Token authentication.

4.19: Blog List Expansion, step 7

Modify adding new blogs so that it is only possible if a valid token is sent with the HTTP POST request. The user identified by the token is designated as the creator of the blog.

4.20*: Blog List Expansion, step 8

This example from part 4 shows taking the token from the header with the `getTokenFrom` helper function in *controllers/blogs.js*.

If you used the same solution, refactor taking the token to a middleware. The middleware should take the token from the *Authorization* header and assign it to the *token* field of the *request* object.

In other words, if you register this middleware in the *app.js* file before all routes

```
app.use(middleware.tokenExtractor)
```

copy

Routes can access the token with `request.token`:

```
blogsRouter.post('/', async (request, response) => {
  // ..
  const decodedToken = jwt.verify(request.token, process.env.SECRET)
  // ..
})
```

Remember that a normal middleware function is a function with three parameters, that at the end calls the last parameter *next* to move the control to the next middleware:

```
const tokenExtractor = (request, response, next) => {
  // code that extracts the token

  next()
}
```

4.21*: Blog List Expansion, step 9

Change the delete blog operation so that a blog can be deleted only by the user who added it. Therefore, deleting a blog is possible only if the token sent with the request is the same as that of the blog's creator.

If deleting a blog is attempted without a token or by an invalid user, the operation should return a suitable status code.

Note that if you fetch a blog from the database,

```
const blog = await Blog.findById(...)
```

the field *blog.user* does not contain a string, but an object. So if you want to compare the ID of the object fetched from the database and a string ID, a normal comparison operation does not work. The ID fetched from the database must be parsed into a string first.

```
if ( blog.user.toString() === userid.toString() ) ...
```

4.22*: Blog List Expansion, step 10

Both the new blog creation and blog deletion need to find out the identity of the user who is doing the operation. The middleware `tokenExtractor` that we did in exercise 4.20 helps but still both the handlers of *post* and *delete* operations need to find out who the user holding a specific token is.

Now create a new middleware `userExtractor`, that finds out the user and sets it to the request object. When you register the middleware in *app.js*

```
app.use(middleware.userExtractor)
```

the user will be set in the field `request.user` :

```
blogsRouter.post('/', async (request, response) => {
  // get user from request object
  const user = request.user
  // ..
})

blogsRouter.delete('/:id', async (request, response) => {
  // get user from request object
  const user = request.user
  // ..
})
```

Note that it is possible to register a middleware only for a specific set of routes. So instead of using `userExtractor` with all the routes,

```
const middleware = require('../utils/middleware');
// ...

// use the middleware in all routes
app.use(middleware.userExtractor)

app.use('/api/blogs', blogsRouter)
app.use('/api/users', usersRouter)
app.use('/api/login', loginRouter)
```

we could register it to be only executed with path */api/blogs* routes:

```
const middleware = require('../utils/middleware');
// ...

// use the middleware only in /api/blogs routes
app.use('/api/blogs', middleware.userExtractor, blogsRouter)
app.use('/api/users', usersRouter)
app.use('/api/login', loginRouter)
```

As can be seen, this happens by chaining multiple middlewares as the arguments of the function *use*. It would also be possible to register a middleware only for a specific operation:

```
const middleware = require('../utils/middleware');
// ...

router.post('/', middleware.userExtractor, async (request, response) => {
  // ...
}
```

4.23*: Blog List Expansion, step 11

After adding token-based authentication the tests for adding a new blog broke down. Fix them. Also, write a new test to ensure adding a blog fails with the proper status code *401 Unauthorized* if a token is not provided.

This is most likely useful when doing the fix.

This is the last exercise for this part of the course and it's time to push your code to GitHub and mark all of your finished exercises to the exercise submission system.

Propose changes to material

Part 4c
Previous part

Part 4e
Next part

About course

Course contents

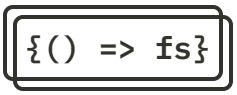
FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON



e Legacy: Testing with Jest

This is the old (pre 13th February 2024) content on testing that is using Jest as the testing library. You may continue using this material if you have already started writing tests with Jest. In other case, you should ignore this page.

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 reverse = (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 = {
  reverse,
  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`, which 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
```

copy

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 ../frontend/ && npm run build && cp -r build
../backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs",
    "lint": "eslint .",
    "test": "jest --verbose"
  },
  //...
}
```

copy

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"
 }
}
```

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

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

test('reverse of a', () => {
  const result = reverse('a')

  expect(result).toBe('a')
})

test('reverse of react', () => {
  const result = reverse('react')

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

test('reverse of releveler', () => {
  const result = reverse('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,
  },
  // ...
}
```

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

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

copy

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 = reverse('react')

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

copy

First, we execute the code to be tested, meaning that we generate a reverse 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:

```
→  noteapp git:(part4-2) ✗ npm test

> noteapp@1.0.0 test
> jest --verbose

 PASS  tests/reverse.test.js
  ✓ reverse of a (2 ms)
  ✓ reverse of react (1 ms)
  ✓ reverse of saippuakauppias

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.804 s
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('reverse of react', () => {
  const result = reverse('react')
```

copy

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

Running this test results in the following error message:

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:

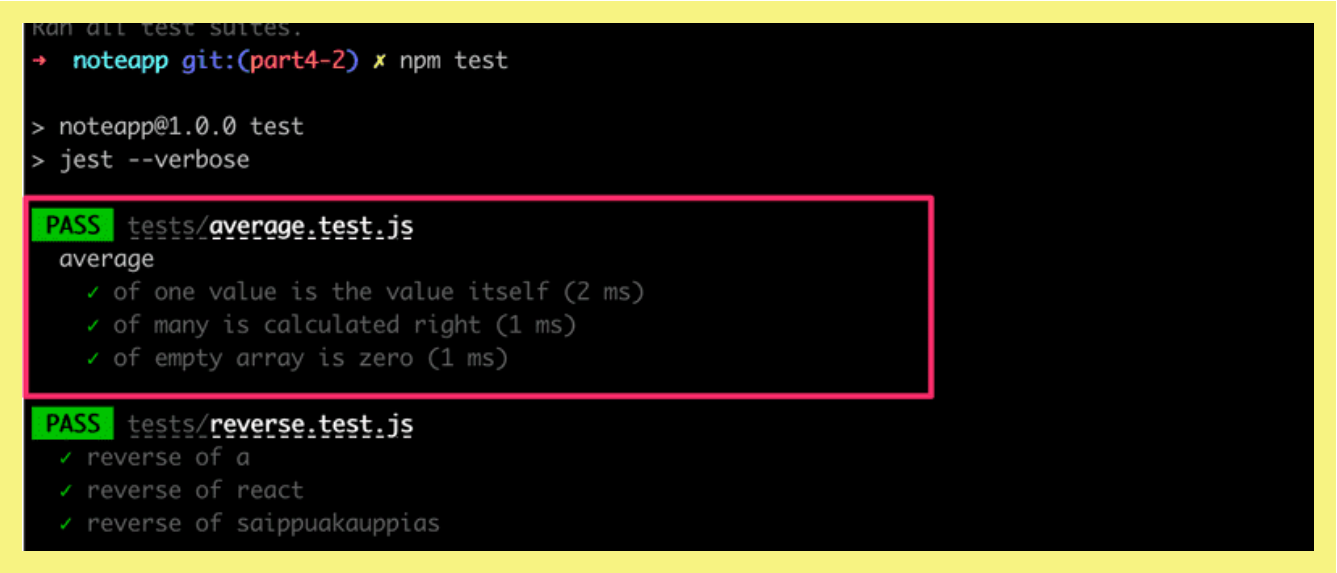
```
const average = array => {
  const reducer = (sum, item) => {
    return sum + item
  }

  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 were 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:



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 in 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, step 1

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
}
```

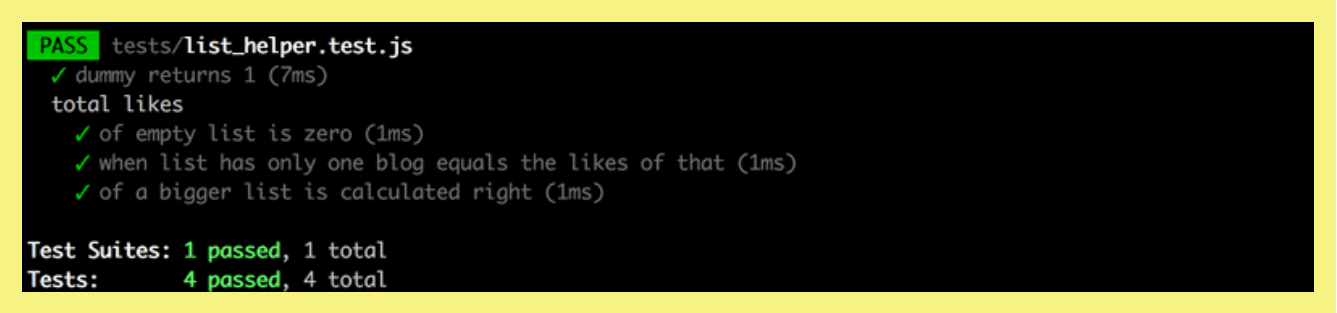
Verify that your test configuration works with the following test:

```
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, step 2

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:



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: 'https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf',
      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'
```

copy

4.5*: Helper Functions and Unit Tests, step 3

Define a new `favoriteBlog` function that receives a list of blogs as a parameter. The function finds out which blog has the 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
}
```

copy

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, step 4

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

```
{
  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, step 5

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 number 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.

We will now start writing tests for the backend. Since the backend does not contain any complicated logic, it doesn't make sense to write unit tests for it. The only potential thing we could unit test is the `toJSON` method that is used for formatting notes.

In some situations, it can be beneficial to implement some of the backend tests by mocking the database instead of using a real database. One library that could be used for this is mongodb-memory-server.

Since our application's backend is still relatively simple, we will decide to test the entire application through its REST API, so that the database is also included. This kind of testing where multiple components of the system are being tested as a group is called integration testing.

Test environment

In one of the previous chapters of the course material, we mentioned that when your backend server is running in Fly.io or Render, it is in *production* mode.

The convention in Node is to define the execution mode of the application with the *NODE_ENV* environment variable. In our current application, we only load the environment variables defined in the *.env* file if the application is *not* in production mode.

It is common practice to define separate modes for development and testing.

Next, let's change the scripts in our notes application *package.json* file, so that when tests are run, *NODE_ENV* gets the value *test*:

```
{
  // ...
  "scripts": {
    "start": "NODE_ENV=production node index.js",
    "dev": "NODE_ENV=development nodemon index.js",
    "build:ui": "rm -rf build && cd ../frontend/ && npm run build && cp -r build
../backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs",
    "lint": "eslint .",
    "test": "NODE_ENV=test jest --verbose --runInBand"
  },
  // ...
}
```

We also added the runInBand option to the npm script that executes the tests. This option will prevent Jest from running tests in parallel; we will discuss its significance once our tests start using the database.

We specified the mode of the application to be *development* in the `npm run dev` script that uses nodemon. We also specified that the default `npm start` command will define the mode as *production*.

There is a slight issue in the way that we have specified the mode of the application in our scripts: it will not work on Windows. We can correct this by installing the cross-env package as a development dependency with the command:

```
npm install --save-dev cross-env
```

We can then achieve cross-platform compatibility by using the cross-env library in our npm scripts defined in *package.json*:

```
{
  // ...
  "scripts": {
    "start": "cross-env NODE_ENV=production node index.js",
    "dev": "cross-env NODE_ENV=development nodemon index.js",
    // ...
    "test": "cross-env NODE_ENV=test jest --verbose --runInBand",
  },
```

```
  // ...
}
```

NB: If you are deploying this application to Fly.io/Render, keep in mind that if cross-env is saved as a development dependency, it would cause an application error on your web server. To fix this, change cross-env to a production dependency by running this in the command line:

```
npm install cross-env
```

copy

Now we can modify the way that our application runs in different modes. As an example of this, we could define the application to use a separate test database when it is running tests.

We can create our separate test database in MongoDB Atlas. This is not an optimal solution in situations where many people are developing the same application. Test execution in particular typically requires a single database instance that is not used by tests that are running concurrently.

It would be better to run our tests using a database that is installed and running on the developer's local machine. The optimal solution would be to have every test execution use a separate database. This is "relatively simple" to achieve by running Mongo in-memory or by using Docker containers. We will not complicate things and will instead continue to use the MongoDB Atlas database.

Let's make some changes to the module that defines the application's configuration in `utils/config.js`:

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

const PORT = process.env.PORT

const MONGODB_URI = process.env.NODE_ENV === 'test'
  ? process.env.TEST_MONGODB_URI
  : process.env.MONGODB_URI

module.exports = {
  MONGODB_URI,
  PORT
}
```

copy

The *.env* file has *separate variables* for the database addresses of the development and test databases:

```
MONGODB_URI=mongodb+srv://fullstack:thepasswordishere@cluster0.o1opl.mongodb.net/noteApp?
retryWrites=true&w=majority
PORT=3001

TEST_MONGODB_URI=mongodb+srv://fullstack:thepasswordishere@cluster0.o1opl.mongodb.net/testNo
retryWrites=true&w=majority
```

copy

The `config` module that we have implemented slightly resembles the node-config package. Writing our implementation is justified since our application is simple, and also because it teaches us valuable lessons.

These are the only changes we need to make to our application's code.

You can find the code for our current application in its entirety in the *part4-2* branch of this GitHub repository.

supertest

Let's use the supertest package to help us write our tests for testing the API.

We will install the package as a development dependency:

```
npm install --save-dev supertest
```

copy

Let's write our first test in the *tests/note_api.test.js* file:

```
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')

const api = supertest(app)

test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
})

afterAll(async () => {
  await mongoose.connection.close()
})
```

copy

The test imports the Express application from the *app.js* module and wraps it with the *supertest* function into a so-called superagent object. This object is assigned to the *api* variable and tests can use it for making HTTP requests to the backend.

Our test makes an HTTP GET request to the *api/notes* url and verifies that the request is responded to with the status code 200. The test also verifies that the *Content-Type* header is set to *application/json*, indicating that the data is in the desired format.

Checking the value of the header uses a bit strange looking syntax:

```
.expect('Content-Type', /application\/json/)
```

copy

The desired value is now defined as regular expression or in short regex. The regex starts and ends with a slash /, because the desired string *application/json* also contains the same slash, it is preceded by a \ so that it is not interpreted as a regex termination character.

In principle, the test could also have been defined as a string

```
.expect('Content-Type', 'application/json')
```

copy

The problem here, however, is that when using a string, the value of the header must be exactly the same. For the regex we defined, it is acceptable that the header *contains* the string in question. The actual value of the header is *application/json; charset=utf-8*, i.e. it also contains information about character encoding. However, our test is not interested in this and therefore it is better to define the test as a regex instead of an exact string.

The test contains some details that we will explore a bit later on. The arrow function that defines the test is preceded by the *async* keyword and the method call for the *api* object is preceded by the *await* keyword. We will write a few tests and then take a closer look at this async/await magic. Do not concern yourself with them for now, just be assured that the example tests work correctly. The async/await syntax is related to the fact that making a request to the API is an *asynchronous* operation. The async/await syntax can be used for writing asynchronous code with the appearance of synchronous code.

Once all the tests (there is currently only one) have finished running we have to close the database connection used by Mongoose. This can be easily achieved with the afterAll method:

```
afterAll(async () => {
  await mongoose.connection.close()
})
```

copy

When running your tests you may run across the following console warning:

```
Jest did not exit one second after the test run has completed.

This usually means that there are asynchronous operations that weren't stopped in your tests. Consider running Jes
t with `--detectOpenHandles` to troubleshoot this issue.
```

The problem is quite likely caused by the Mongoose version 6.x, the problem does not appear when version 5.x or 7.x is used. Mongoose documentation does not recommend testing Mongoose applications with Jest.

One way to get rid of this is to add to the directory *tests* a file *teardown.js* with the following content

```
module.exports = () => {
  process.exit(0)
}
```

and by extending the Jest definitions in the *package.json* as follows

```
{
 //...
 "jest": {
   "testEnvironment": "node",
   "globalTeardown": "./tests/teardown.js"
 }
}
```

Another error you may come across is your test takes longer than the default Jest test timeout of 5000 ms. This can be solved by adding a third parameter to the test function:

```
test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
}, 100000)
```

This third parameter sets the timeout to 100000 ms. A long timeout ensures that our test won't fail due to the time it takes to run. (A long timeout may not be what you want for tests based on performance or speed, but this is fine for our example tests).

If you still encounter issues with mongoose timeouts, set `bufferTimeoutMS` variable to a value significantly higher than 10000 (10 seconds). You could set it like this at the top, right after the `require` statements. `mongoose.set("bufferTimeoutMS", 30000)`

One tiny but important detail: at the beginning of this part we extracted the Express application into the *app.js* file, and the role of the *index.js* file was changed to launch the application at the specified port via `app.listen`:

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

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

The tests only use the Express application defined in the *app.js* file, which does not listen to any ports:

```
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')

const api = supertest(app)

// ...
```

The documentation for supertest says the following:

> *if the server is not already listening for connections then it is bound to an ephemeral port for you so there is no need to keep track of ports.*

In other words, supertest takes care that the application being tested is started at the port that it uses internally.

Let's add two notes to the test database using the `mongo.js` program (here we must remember to switch to the correct database url).

Let's write a few more tests:

```
test('there are two notes', async () => {
  const response = await api.get('/api/notes')

  expect(response.body).toHaveLength(2)
})

test('the first note is about HTTP methods', async () => {
  const response = await api.get('/api/notes')

  expect(response.body[0].content).toBe('HTML is easy')
})
```

Both tests store the response of the request to the `response` variable, and unlike the previous test that used the methods provided by `supertest` for verifying the status code and headers, this time we are inspecting the response data stored in *response.body* property. Our tests verify the format and content of the response data with the `expect` method of Jest.

The benefit of using the async/await syntax is starting to become evident. Normally we would have to use callback functions to access the data returned by promises, but with the new syntax things are a lot more comfortable:

```
const response = await api.get('/api/notes')
```

```
// execution gets here only after the HTTP request is complete
// the result of HTTP request is saved in variable response
expect(response.body).toHaveLength(2)
```

The middleware that outputs information about the HTTP requests is obstructing the test execution output. Let us modify the logger so that it does not print to the console in test mode:

```
const info = (...params) => {
  if (process.env.NODE_ENV !== 'test') {
    console.log(...params)
  }
}

const error = (...params) => {
  if (process.env.NODE_ENV !== 'test') {
    console.error(...params)
  }
}

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

Initializing the database before tests

Testing appears to be easy and our tests are currently passing. However, our tests are bad as they are dependent on the state of the database, that now happens to have two notes. To make them more robust, we have to reset the database and generate the needed test data in a controlled manner before we run the tests.

Our tests are already using the afterAll function of Jest to close the connection to the database after the tests are finished executing. Jest offers many other functions that can be used for executing operations once before any test is run or every time before a test is run.

Let's initialize the database *before every test* with the beforeEach function:

```
const mongoose = require('mongoose')
const supertest = require('supertest')
const app = require('../app')
const api = supertest(app)
const Note = require('../models/note')

const initialNotes = [
  {
    content: 'HTML is easy',
    important: false,
  },
  {
    content: 'Browser can execute only JavaScript',
```

```
    important: true,
  },
]

beforeEach(async () => {
  await Note.deleteMany({})
  let noteObject = new Note(initialNotes[0])
  await noteObject.save()
  noteObject = new Note(initialNotes[1])
  await noteObject.save()
})
// ...
```

The database is cleared out at the beginning, and after that, we save the two notes stored in the `initialNotes` array to the database. By doing this, we ensure that the database is in the same state before every test is run.

Let's also make the following changes to the last two tests:

```
test('all notes are returned', async () => {
  const response = await api.get('/api/notes')

  expect(response.body).toHaveLength(initialNotes.length)
})

test('a specific note is within the returned notes', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)
  expect(contents).toContain(
    'Browser can execute only JavaScript'
  )
})
```

Pay special attention to the expect in the latter test. The `response.body.map(r => r.content)` command is used to create an array containing the content of every note returned by the API. The `toContain` method is used for checking that the note given to it as a parameter is in the list of notes returned by the API.

Running tests one by one

The `npm test` command executes all of the tests for the application. When we are writing tests, it is usually wise to only execute one or two tests. Jest offers a few different ways of accomplishing this, one of which is the `only` method. If tests are written across many files, this method is not great.

A better option is to specify the tests that need to be run as parameters of the *npm test* command.

The following command only runs the tests found in the *tests/note_api.test.js* file:

```
npm test -- tests/note_api.test.js
```

copy

The *-t* option can be used for running tests with a specific name:

```
npm test -- -t "a specific note is within the returned notes"
```

copy

The provided parameter can refer to the name of the test or the describe block. The parameter can also contain just a part of the name. The following command will run all of the tests that contain *notes* in their name:

```
npm test -- -t 'notes'
```

copy

NB: When running a single test, the mongoose connection might stay open if no tests using the connection are run. The problem might be because supertest primes the connection, but Jest does not run the afterAll portion of the code.

async/await

Before we write more tests let's take a look at the `async` and `await` keywords.

The async/await syntax that was introduced in ES7 makes it possible to use *asynchronous functions that return a promise* in a way that makes the code look synchronous.

As an example, the fetching of notes from the database with promises looks like this:

```
Note.find({}).then(notes => {
  console.log('operation returned the following notes', notes)
})
```

copy

The `Note.find()` method returns a promise and we can access the result of the operation by registering a callback function with the `then` method.

All of the code we want to execute once the operation finishes is written in the callback function. If we wanted to make several asynchronous function calls in sequence, the situation would soon become painful. The asynchronous calls would have to be made in the callback. This would likely lead to complicated code and could potentially give birth to a so-called callback hell.

By chaining promises we could keep the situation somewhat under control, and avoid callback hell by creating a fairly clean chain of `then` method calls. We have seen a few of these during the course. To

illustrate this, you can view an artificial example of a function that fetches all notes and then deletes the first one:

```
Note.find({})
  .then(notes => {
    return notes[0].deleteOne()
  })
  .then(response => {
    console.log('the first note is removed')
    // more code here
  })
```

The then-chain is alright, but we can do better. The generator functions introduced in ES6 provided a clever way of writing asynchronous code in a way that "looks synchronous". The syntax is a bit clunky and not widely used.

The `async` and `await` keywords introduced in ES7 bring the same functionality as the generators, but in an understandable and syntactically cleaner way to the hands of all citizens of the JavaScript world.

We could fetch all of the notes in the database by utilizing the await operator like this:

```
const notes = await Note.find({})

console.log('operation returned the following notes', notes)
```

The code looks exactly like synchronous code. The execution of code pauses at `const notes = await Note.find({})` and waits until the related promise is *fulfilled*, and then continues its execution to the next line. When the execution continues, the result of the operation that returned a promise is assigned to the `notes` variable.

The slightly complicated example presented above could be implemented by using await like this:

```
const notes = await Note.find({})
const response = await notes[0].deleteOne()

console.log('the first note is removed')
```

Thanks to the new syntax, the code is a lot simpler than the previous then-chain.

There are a few important details to pay attention to when using async/await syntax. To use the await operator with asynchronous operations, they have to return a promise. This is not a problem as such, as regular asynchronous functions using callbacks are easy to wrap around promises.

The await keyword can't be used just anywhere in JavaScript code. Using await is possible only inside of an async function.

This means that in order for the previous examples to work, they have to be using async functions. Notice the first line in the arrow function definition:

```
const main = async () => {
  const notes = await Note.find({})
  console.log('operation returned the following notes', notes)

  const response = await notes[0].deleteOne()
  console.log('the first note is removed')
}

main()
```

The code declares that the function assigned to `main` is asynchronous. After this, the code calls the function with `main()`.

async/await in the backend

Let's start to change the backend to async and await. As all of the asynchronous operations are currently done inside of a function, it is enough to change the route handler functions into async functions.

The route for fetching all notes gets changed to the following:

```
notesRouter.get('/', async (request, response) => {
  const notes = await Note.find({})
  response.json(notes)
})
```

We can verify that our refactoring was successful by testing the endpoint through the browser and by running the tests that we wrote earlier.

You can find the code for our current application in its entirety in the *part4-3* branch of this GitHub repository.

More tests and refactoring the backend

When code gets refactored, there is always the risk of regression, meaning that existing functionality may break. Let's refactor the remaining operations by first writing a test for each route of the API.

Let's start with the operation for adding a new note. Let's write a test that adds a new note and verifies that the number of notes returned by the API increases and that the newly added note is in the list.

```
test('a valid note can be added', async () => {
  const newNote = {
    content: 'async/await simplifies making async calls',
    important: true,
  }

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(201)
    .expect('Content-Type', /application\/json/)

  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)

  expect(response.body).toHaveLength(initialNotes.length + 1)
  expect(contents).toContain(
    'async/await simplifies making async calls'
  )
})
```

Test fails since we are by accident returning the status code *200 OK* when a new note is created. Let us change that to *201 CREATED*:

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

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

  note.save()
    .then(savedNote => {
      response.status(201).json(savedNote)
    })
    .catch(error => next(error))
})
```

Let's also write a test that verifies that a note without content will not be saved into the database.

```
test('note without content is not added', async () => {
  const newNote = {
    important: true
  }

  await api
```

```
    .post('/api/notes')
    .send(newNote)
    .expect(400)

  const response = await api.get('/api/notes')

  expect(response.body).toHaveLength(initialNotes.length)
})
```

Both tests check the state stored in the database after the saving operation, by fetching all the notes of the application.

```
const response = await api.get('/api/notes')
```

copy

The same verification steps will repeat in other tests later on, and it is a good idea to extract these steps into helper functions. Let's add the function into a new file called *tests/test_helper.js* which is in the same directory as the test file.

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

const initialNotes = [
  {
    content: 'HTML is easy',
    important: false
  },
  {
    content: 'Browser can execute only JavaScript',
    important: true
  }
]

const nonExistingId = async () => {
  const note = new Note({ content: 'willremovethissoon' })
  await note.save()
  await note.deleteOne()

  return note._id.toString()
}

const notesInDb = async () => {
  const notes = await Note.find({})
  return notes.map(note => note.toJSON())
}

module.exports = {
  initialNotes, nonExistingId, notesInDb
}
```

copy

The module defines the `notesInDb` function that can be used for checking the notes stored in the database. The `initialNotes` array containing the initial database state is also in the module. We also define the `nonExistingId` function ahead of time, which can be used for creating a database object ID that does not belong to any note object in the database.

Our tests can now use the helper module and be changed like this:

```
const supertest = require('supertest')
const mongoose = require('mongoose')
const helper = require('./test_helper')
const app = require('../app')
const api = supertest(app)

const Note = require('../models/note')

beforeEach(async () => {
  await Note.deleteMany({})

  let noteObject = new Note(helper.initialNotes[0])
  await noteObject.save()

  noteObject = new Note(helper.initialNotes[1])
  await noteObject.save()
})

test('notes are returned as json', async () => {
  await api
    .get('/api/notes')
    .expect(200)
    .expect('Content-Type', /application\/json/)
})

test('all notes are returned', async () => {
  const response = await api.get('/api/notes')

  expect(response.body).toHaveLength(helper.initialNotes.length)
})

test('a specific note is within the returned notes', async () => {
  const response = await api.get('/api/notes')

  const contents = response.body.map(r => r.content)

  expect(contents).toContain(
    'Browser can execute only JavaScript'
  )
})

test('a valid note can be added ', async () => {
  const newNote = {
    content: 'async/await simplifies making async calls',
    important: true,
```

```
  }

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(201)
    .expect('Content-Type', /application\/json/)

  const notesAtEnd = await helper.notesInDb()
  expect(notesAtEnd).toHaveLength(helper.initialNotes.length + 1)

  const contents = notesAtEnd.map(n => n.content)
  expect(contents).toContain(
    'async/await simplifies making async calls'
  )
})

test('note without content is not added', async () => {
  const newNote = {
    important: true
  }

  await api
    .post('/api/notes')
    .send(newNote)
    .expect(400)

  const notesAtEnd = await helper.notesInDb()

  expect(notesAtEnd).toHaveLength(helper.initialNotes.length)
})

afterAll(async () => {
  await mongoose.connection.close()
})
```

The code using promises works and the tests pass. We are ready to refactor our code to use the async/await syntax.

We make the following changes to the code that takes care of adding a new note (notice that the route handler definition is preceded by the `async` keyword):

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

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

  const savedNote = await note.save()
```

```
  response.status(201).json(savedNote)
})
```

There's a slight problem with our code: we don't handle error situations. How should we deal with them?

Error handling and async/await

If there's an exception while handling the POST request we end up in a familiar situation:

```
Method: POST
Path:   /api/notes
Body:   { important: true }
---
(node:89372) UnhandledPromiseRejectionWarning: ValidationError: Note validation failed: content: Path `content` is
 required.
    at new ValidationError (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/
lib/error/validation.js:30:11)
    at model.Document.invalidate (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mon
goose/lib/document.js:2071:32)
    at p.doValidate.skipSchemaValidators (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_mod
ules/mongoose/lib/document.js:1934:17)
    at /Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/node_modules/mongoose/lib/schematype.js:929
:9
    at process._tickCallback (internal/process/next_tick.js:172:11)
(node:89372) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwi
```

In other words, we end up with an unhandled promise rejection, and the request never receives a response.

With async/await the recommended way of dealing with exceptions is the old and familiar `try/catch` mechanism:

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

  const note = new Note({
    content: body.content,
    important: body.important || false,
  })
  try {
    const savedNote = await note.save()
    response.status(201).json(savedNote)
  } catch(exception) {
    next(exception)
  }
})
```

The catch block simply calls the `next` function, which passes the request handling to the error handling middleware.

After making the change, all of our tests will pass once again.

Next, let's write tests for fetching and removing an individual note:

```
test('a specific note can be viewed', async () => {
  const notesAtStart = await helper.notesInDb()

  const noteToView = notesAtStart[0]

  const resultNote = await api
    .get(`/api/notes/${noteToView.id}`)
    .expect(200)
    .expect('Content-Type', /application\/json/)

  expect(resultNote.body).toEqual(noteToView)
})

test('a note can be deleted', async () => {
  const notesAtStart = await helper.notesInDb()
  const noteToDelete = notesAtStart[0]

  await api
    .delete(`/api/notes/${noteToDelete.id}`)
    .expect(204)

  const notesAtEnd = await helper.notesInDb()

  expect(notesAtEnd).toHaveLength(
    helper.initialNotes.length - 1
  )

  const contents = notesAtEnd.map(r => r.content)

  expect(contents).not.toContain(noteToDelete.content)
})
```

Both tests share a similar structure. In the initialization phase, they fetch a note from the database. After this, the tests call the actual operation being tested, which is highlighted in the code block. Lastly, the tests verify that the outcome of the operation is as expected.

The tests pass and we can safely refactor the tested routes to use async/await:

```
notesRouter.get('/:id', async (request, response, next) => {
  try {
    const note = await Note.findById(request.params.id)
    if (note) {
      response.json(note)
    } else {
      response.status(404).end()
    }
  } catch(exception) {
    next(exception)
```

```
  }
})

notesRouter.delete('/:id', async (request, response, next) => {
  try {
    await Note.findByIdAndDelete(request.params.id)
    response.status(204).end()
  } catch(exception) {
    next(exception)
  }
})
```

You can find the code for our current application in its entirety in the *part4-4* branch of this GitHub repository.

Eliminating the try-catch

Async/await unclutters the code a bit, but the 'price' is the *try/catch* structure required for catching exceptions. All of the route handlers follow the same structure

```
try {
  // do the async operations here
} catch(exception) {
  next(exception)
}
```

One starts to wonder if it would be possible to refactor the code to eliminate the *catch* from the methods?

The express-async-errors library has a solution for this.

Let's install the library

```
npm install express-async-errors
```

Using the library is *very* easy. You introduce the library in *app.js*, `before` you import your routes:

```
const config = require('./utils/config')
const express = require('express')
require('express-async-errors')
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')
```

```
// ...

module.exports = app
```

The 'magic' of the library allows us to eliminate the try-catch blocks completely. For example the route for deleting a note

```
notesRouter.delete('/:id', async (request, response, next) => {
  try {
    await Note.findByIdAndDelete(request.params.id)
    response.status(204).end()
  } catch (exception) {
    next(exception)
  }
})
```

becomes

```
notesRouter.delete('/:id', async (request, response) => {
  await Note.findByIdAndDelete(request.params.id)
  response.status(204).end()
})
```

Because of the library, we do not need the `next(exception)` call anymore. The library handles everything under the hood. If an exception occurs in an *async* route, the execution is automatically passed to the error handling middleware.

The other routes become:

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

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

  const savedNote = await note.save()
  response.status(201).json(savedNote)
})

notesRouter.get('/:id', async (request, response) => {
  const note = await Note.findById(request.params.id)
  if (note) {
    response.json(note)
  } else {
```

```
    response.status(404).end()
  }
})
```

Optimizing the beforeEach function

Let's return to writing our tests and take a closer look at the `beforeEach` function that sets up the tests:

```
beforeEach(async () => {
  await Note.deleteMany({})

  let noteObject = new Note(helper.initialNotes[0])
  await noteObject.save()

  noteObject = new Note(helper.initialNotes[1])
  await noteObject.save()
})
```

The function saves the first two notes from the `helper.initialNotes` array into the database with two separate operations. The solution is alright, but there's a better way of saving multiple objects to the database:

```
beforeEach(async () => {
  await Note.deleteMany({})
  console.log('cleared')

  helper.initialNotes.forEach(async (note) => {
    let noteObject = new Note(note)
    await noteObject.save()
    console.log('saved')
  })
  console.log('done')
})

test('notes are returned as json', async () => {
  console.log('entered test')
  // ...
}
```

We save the notes stored in the array into the database inside of a `forEach` loop. The tests don't quite seem to work however, so we have added some console logs to help us find the problem.

The console displays the following output:

```
cleared
done
entered test
saved
saved
```

Despite our use of the async/await syntax, our solution does not work as we expected it to. The test execution begins before the database is initialized!

The problem is that every iteration of the forEach loop generates an asynchronous operation, and `beforeEach` won't wait for them to finish executing. In other words, the `await` commands defined inside of the `forEach` loop are not in the `beforeEach` function, but in separate functions that `beforeEach` will not wait for.

Since the execution of tests begins immediately after `beforeEach` has finished executing, the execution of tests begins before the database state is initialized.

One way of fixing this is to wait for all of the asynchronous operations to finish executing with the Promise.all method:

```
beforeEach(async () => {
  await Note.deleteMany({})

  const noteObjects = helper.initialNotes
    .map(note => new Note(note))
  const promiseArray = noteObjects.map(note => note.save())
  await Promise.all(promiseArray)
})
```

The solution is quite advanced despite its compact appearance. The `noteObjects` variable is assigned to an array of Mongoose objects that are created with the `Note` constructor for each of the notes in the `helper.initialNotes` array. The next line of code creates a new array that *consists of promises*, that are created by calling the `save` method of each item in the `noteObjects` array. In other words, it is an array of promises for saving each of the items to the database.

The Promise.all method can be used for transforming an array of promises into a single promise, that will be *fulfilled* once every promise in the array passed to it as a parameter is resolved. The last line of code `await Promise.all(promiseArray)` waits until every promise for saving a note is finished, meaning that the database has been initialized.

> The returned values of each promise in the array can still be accessed when using the Promise.all method. If we wait for the promises to be resolved with the `await` syntax `const results = await Promise.all(promiseArray)`, the operation will return an array that contains the resolved values for each promise in the `promiseArray`, and they appear in the same order as the promises in the array.

Promise.all executes the promises it receives in parallel. If the promises need to be executed in a particular order, this will be problematic. In situations like this, the operations can be executed inside of a `for...of` block, that guarantees a specific execution order.

```
beforeEach(async () => {
  await Note.deleteMany({})

  for (let note of helper.initialNotes) {
    let noteObject = new Note(note)
    await noteObject.save()
  }
})
```

The asynchronous nature of JavaScript can lead to surprising behavior, and for this reason, it is important to pay careful attention when using the async/await syntax. Even though the syntax makes it easier to deal with promises, it is still necessary to understand how promises work!

The code for our application can be found on GitHub, branch *part4-5*.

A true full stack developer's oath

Making tests brings yet another layer of challenge to programming. We have to update our full stack developer oath to remind you that systematicity is also key when developing tests.

So we should once more extend our oath:

Full stack development is *extremely hard*, that is why I will use all the possible means to make it easier

- I will have my browser developer console open all the time
- I will use the network tab of the browser dev tools to ensure that frontend and backend are communicating as I expect
- I will constantly keep an eye on the state of the server to make sure that the data sent there by the frontend is saved as I expect
- I will keep an eye on the database: does the backend save data there in the right format
- I will progress in small steps
- *I will write lots of* `console.log` *statements to make sure I understand how the code and the tests behave and to help pinpoint problems*
- If my code does not work, I will not write more code. Instead, I start deleting the code until it works or just return to a state when everything was still working
- *If a test does not pass, I make sure that the tested functionality for sure works in the application*
- When I ask for help in the course Discord or Telegram channel or elsewhere I formulate my questions properly, see here how to ask for help

Exercises 4.8.-4.12.

NB: the material uses the toContain matcher in several places to verify that an array contains a specific element. It's worth noting that the method uses the === operator for comparing and matching elements, which means that it is often not well-suited for matching objects. In most cases, the appropriate method for verifying objects in arrays is the toContainEqual matcher. However, the model solutions don't check for objects in arrays with matchers, so using the method is not required for solving the exercises.

Warning: If you find yourself using async/await and *then* methods in the same code, it is almost guaranteed that you are doing something wrong. Use one or the other and don't mix the two.

4.8: Blog List Tests, step 1

Use the supertest package for writing a test that makes an HTTP GET request to the */api/blogs* URL. Verify that the blog list application returns the correct amount of blog posts in the JSON format.

Once the test is finished, refactor the route handler to use the async/await syntax instead of promises.

Notice that you will have to make similar changes to the code that were made in the material, like defining the test environment so that you can write tests that use separate databases.

NB: When running the tests, you may run into the following warning:

```
  console.warn node_modules/mongoose/lib/helpers/printJestWarning.js:4
    Mongoose: looks like you're trying to test a Mongoose app with Jest's default jsdom test environment. Please m
ake sure you read Mongoose's docs on configuring Jest to test Node.js apps: http://mongoosejs.com/docs/jest.html
```

One way to get rid of this is to add to the *tests* directory a file *teardown.js* with the following content

```
module.exports = () => {
  process.exit(0)
}
```

and by extending the Jest definitions in the *package.json* as follows

```
{
 //...
 "jest": {
   "testEnvironment": "node",
   "globalTeardown": "./tests/teardown.js"
```

```
  }
}
```

NB: when you are writing your tests ***it is better to not execute them all***, only execute the ones you are working on. Read more about this here.

4.9: Blog List Tests, step 2

Write a test that verifies that the unique identifier property of the blog posts is named *id*, by default the database names the property *_id*. Verifying the existence of a property is easily done with Jest's toBeDefined matcher.

Make the required changes to the code so that it passes the test. The toJSON method discussed in part 3 is an appropriate place for defining the *id* parameter.

4.10: Blog List Tests, step 3

Write a test that verifies that making an HTTP POST request to the */api/blogs* URL successfully creates a new blog post. At the very least, verify that the total number of blogs in the system is increased by one. You can also verify that the content of the blog post is saved correctly to the database.

Once the test is finished, refactor the operation to use async/await instead of promises.

4.11*: Blog List Tests, step 4

Write a test that verifies that if the *likes* property is missing from the request, it will default to the value 0. Do not test the other properties of the created blogs yet.

Make the required changes to the code so that it passes the test.

4.12*: Blog List tests, step 5

Write tests related to creating new blogs via the */api/blogs* endpoint, that verify that if the *title* or *url* properties are missing from the request data, the backend responds to the request with the status code *400 Bad Request*.

Make the required changes to the code so that it passes the test.

Refactoring tests

Our test coverage is currently lacking. Some requests like *GET /api/notes/:id* and *DELETE /api/notes/:id* aren't tested when the request is sent with an invalid id. The grouping and organization of tests could also use some improvement, as all tests exist on the same "top level" in the test file. The readability of the test would improve if we group related tests with *describe* blocks.

Below is an example of the test file after making some minor improvements:

```
const supertest = require('supertest')
const mongoose = require('mongoose')
const helper = require('./test_helper')
const app = require('../app')
const api = supertest(app)

const Note = require('../models/note')

beforeEach(async () => {
  await Note.deleteMany({})
  await Note.insertMany(helper.initialNotes)
})

describe('when there is initially some notes saved', () => {
  test('notes are returned as json', async () => {
    await api
      .get('/api/notes')
      .expect(200)
      .expect('Content-Type', /application\/json/)
  })

  test('all notes are returned', async () => {
    const response = await api.get('/api/notes')

    expect(response.body).toHaveLength(helper.initialNotes.length)
  })

  test('a specific note is within the returned notes', async () => {
    const response = await api.get('/api/notes')

    const contents = response.body.map(r => r.content)

    expect(contents).toContain(
      'Browser can execute only JavaScript'
    )
  })
})

describe('viewing a specific note', () => {
  test('succeeds with a valid id', async () => {
    const notesAtStart = await helper.notesInDb()

    const noteToView = notesAtStart[0]

    const resultNote = await api
      .get(`/api/notes/${noteToView.id}`)
      .expect(200)
      .expect('Content-Type', /application\/json/)

    expect(resultNote.body).toEqual(noteToView)
  })

  test('fails with statuscode 404 if note does not exist', async () => {
    const validNonexistingId = await helper.nonExistingId()
```

```
    await api
      .get(`/api/notes/${validNonexistingId}`)
      .expect(404)
  })

  test('fails with statuscode 400 if id is invalid', async () => {
    const invalidId = '5a3d5da59070081a82a3445'

    await api
      .get(`/api/notes/${invalidId}`)
      .expect(400)
  })
})

describe('addition of a new note', () => {
  test('succeeds with valid data', async () => {
    const newNote = {
      content: 'async/await simplifies making async calls',
      important: true,
    }

    await api
      .post('/api/notes')
      .send(newNote)
      .expect(201)
      .expect('Content-Type', /application\/json/)

    const notesAtEnd = await helper.notesInDb()
    expect(notesAtEnd).toHaveLength(helper.initialNotes.length + 1)

    const contents = notesAtEnd.map(n => n.content)
    expect(contents).toContain(
      'async/await simplifies making async calls'
    )
  })

  test('fails with status code 400 if data invalid', async () => {
    const newNote = {
      important: true
    }

    await api
      .post('/api/notes')
      .send(newNote)
      .expect(400)

    const notesAtEnd = await helper.notesInDb()

    expect(notesAtEnd).toHaveLength(helper.initialNotes.length)
  })
})

describe('deletion of a note', () => {
```

```
  test('succeeds with status code 204 if id is valid', async () => {
    const notesAtStart = await helper.notesInDb()
    const noteToDelete = notesAtStart[0]

    await api
      .delete(`/api/notes/${noteToDelete.id}`)
      .expect(204)

    const notesAtEnd = await helper.notesInDb()

    expect(notesAtEnd).toHaveLength(
      helper.initialNotes.length - 1
    )

    const contents = notesAtEnd.map(r => r.content)

    expect(contents).not.toContain(noteToDelete.content)
  })
})

afterAll(async () => {
  await mongoose.connection.close()
})
```

The test output in the console is grouped according to the *describe* blocks:

```
PASS tests/note_api.test.js (5.473s)
  when there is initially some notes saved
    ✓ notes are returned as json (1737ms)
    ✓ all notes are returned (180ms)
    ✓ a specific note is within the returned notes (208ms)
    viewing a specifin note
      ✓ succeeds with a valid id (226ms)
      ✓ fails with statuscode 404 if note does not exist (297ms)
      ✓ fails with statuscode 400 id is invalid invalid (114ms)
    addition of a new note
      ✓ succeeds with valid data (243ms)
      ✓ fails with status code 400 if data invaild (168ms)
    deletion of a note
      ✓ succeeds with status code 200 if id is valid (272ms)
```

There is still room for improvement, but it is time to move forward.

This way of testing the API, by making HTTP requests and inspecting the database with Mongoose, is by no means the only nor the best way of conducting API-level integration tests for server applications. There is no universal best way of writing tests, as it all depends on the application being tested and available resources.

You can find the code for our current application in its entirety in the *part4-6* branch of this GitHub repository.

Exercises 4.13.-4.14.

4.13 Blog List Expansions, step 1

Implement functionality for deleting a single blog post resource.

Use the async/await syntax. Follow RESTful conventions when defining the HTTP API.

Implement tests for the functionality.

4.14 Blog List Expansions, step 2

Implement functionality for updating the information of an individual blog post.

Use async/await.

The application mostly needs to update the number of *likes* for a blog post. You can implement this functionality the same way that we implemented updating notes in part 3.

Implement tests for the functionality.

Propose changes to material

Part 4d
Previous part

Part 5
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON