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.o1
opl.mongodb.net/testNoteApp?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
```

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()
})
```

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/)
```

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')
```

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()
})
```

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')

  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
}
```

Intializing the database before tests

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)
})
```

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

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

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
```

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"
```

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.

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

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:

```
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:

terminal showing unhandled promise rejection warning

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
```

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 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 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:

node:test output showing grouped describe blocks

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.