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

В одной из предыдущих глав материала курса мы упоминали, что когда ваш серверный сервер запущен в режиме Fly.io или рендеринга, он находится в режиме *production*.

Соглашение в Node заключается в определении режима выполнения приложения с помощью переменной окружения *NODE_ENV*. В нашем текущем приложении мы загружаем переменные среды, определенные в файле *.env*, только если приложение *не* находится в рабочем режиме.

Обычной практикой является определение отдельных режимов для разработки и тестирования.

Далее давайте изменим скрипты в файле *package.json* нашего приложения notes, чтобы при выполнении тестов *NODE_ENV* получал значение *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 .",
  },
  // ...
}
```

Мы указали режим работы приложения для *разработки* в скрипте `npm run dev`, использующем nodemon. Мы также указали, что команда `npm start` по умолчанию будет определять режим как *производственный*.

Есть небольшая проблема в том, как мы указали режим работы приложения в наших скриптах: оно не будет работать в Windows. Мы можем исправить это, установив пакет cross-env в качестве зависимости от разработки с помощью команды:

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

Затем мы можем добиться кроссплатформенной совместимости, используя библиотеку cross-env в наших скриптах npm, определенных в *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: Если вы развертываете это приложение на Fly.io/Render, имейте в виду, что если cross-env сохранен как зависимость для разработки, это приведет к ошибке приложения на вашем веб-сервере. Чтобы исправить это, измените cross-env на производственную зависимость, выполнив в командной строке следующее:

```
npm install cross-env
```

Копировать

Теперь мы можем изменить способ работы нашего приложения в разных режимах. Например, мы можем указать приложению использовать отдельную тестовую базу данных при выполнении тестов.

Мы можем создать нашу отдельную тестовую базу данных в MongoDB Atlas. Это не оптимальное решение в ситуациях, когда много людей разрабатывают одно и то же приложение. Для выполнения тестов, в частности, обычно требуется один экземпляр базы данных, который не используется тестами, выполняемыми одновременно.

Было бы лучше запускать наши тесты, используя базу данных, установленную и работающую на локальном компьютере разработчика. Оптимальным решением было бы, чтобы при выполнении каждого теста использовалась отдельная база данных. Этого "относительно просто" достичь с помощью запуска Mongo в памяти или с помощью Docker контейнеров. Мы не будем усложнять и вместо этого продолжим использовать базу данных MongoDB Atlas.

Давайте внесем некоторые изменения в модуль, который определяет конфигурацию приложения в `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
}
```

Копировать

Файл *.env* содержит *отдельные переменные* для адресов баз данных разработки и тестовых баз данных:

```
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/testNoteApp?retryWrites=true&w=majority
```

Копировать

Реализованный нами модуль `config` немного напоминает пакет node-config. Написание нашей реализации оправдано, поскольку наше приложение простое, а также потому, что оно преподает нам ценные уроки.

Это единственные изменения, которые нам нужно внести в код нашего приложения.

Вы можете найти код для нашего текущего приложения полностью в *части 4-2* в этом репозитории GitHub.

супертест

Давайте воспользуемся пакетом supertest, который поможет нам написать тесты для тестирования API.

Мы установим пакет как зависимость от разработки:

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

Копировать

Давайте запишем наш первый тест в *tests/note_api.test.js* файл:

```
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 () => {
```

Копировать

```
  await mongoose.connection.close()
})
```

Тест импортирует приложение Express из модуля *app.js* и оборачивает его с помощью функции *supertest* в так называемый объект superagent. Этот объект присваивается переменной *api*, и тесты могут использовать его для отправки HTTP-запросов к серверной части.

Наш тест отправляет HTTP GET-запрос к URL-адресу *api / notes* и проверяет, получен ли ответ на запрос с кодом состояния 200. Тест также проверяет, что для заголовка *Content-Type* установлено значение *application /json*, что указывает на то, что данные находятся в желаемом формате.

При проверке значения заголовка используется немного странный синтаксис:

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

Копировать

Желаемое значение теперь определяется как регулярное выражение или сокращенно regex. Регулярное выражение начинается и заканчивается косой чертой /, поскольку требуемая строка *application/json* также содержит такую же косую черту, ей предшествует \, чтобы она не интерпретировалась как символ завершения регулярного выражения.

В принципе, тест также мог быть определен как строка

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

Копировать

Проблема здесь, однако, заключается в том, что при использовании строки значение заголовка должно быть точно таким же. Для определенного нами регулярного выражения допустимо, чтобы заголовок *содержал* рассматриваемую строку. Фактическое значение заголовка - *application/json; charset=utf-8*, т.е. он также содержит информацию о кодировке символов. Однако наш тест этим не интересуется, и поэтому лучше определить тест как регулярное выражение, а не точную строку.

Тест содержит некоторые детали, которые мы рассмотрим чуть позже. Функции arrow, определяющей тест, предшествует ключевое слово *async*, а вызову метода для объекта *api* предшествует ключевое слово *await*. Мы напишем несколько тестов, а затем подробнее рассмотрим эту магию асинхронности / ожидания. Пока не обращайте на них внимания, просто будьте уверены, что примеры тестов работают правильно. Синтаксис async/await связан с тем фактом, что выполнение запроса к API является *асинхронной* операцией. Синтаксис async / await может использоваться для написания асинхронного кода с внешним видом синхронного кода.

После завершения всех тестов (в настоящее время выполняется только один) мы должны закрыть подключение к базе данных, используемое Mongoose. Этого можно легко достичь с помощью метода after:

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

Копировать

Одна крошечная, но важная деталь: в начале этой части мы извлекли приложение Express в *app.js* файл, и роль *index.js* файл был изменен для запуска приложения через указанный порт через `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}`)
})
```

Копировать

В тестах используется только приложение Express, определенное в *app.js* файле, которое не прослушивает никакие порты:

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

const api = supertest(app)

// ...
```

Копировать

В документации к supertest говорится следующее:

> *если сервер еще не прослушивает соединения, то для вас он привязан к временному порту, поэтому нет необходимости отслеживать порты.*

Другими словами, supertest заботится о том, чтобы тестируемое приложение запускалось с порта, который оно использует внутренне.

Давайте добавим два примечания к тестовой базе данных с помощью программы `mongo.js` (здесь мы должны не забыть переключиться на правильный URL базы данных).

Давайте напишем еще несколько тестов:

```
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)
})
```

Оба теста сохраняют ответ на запрос в переменной `response`, и в отличие от предыдущего теста, в котором использовались методы, предоставленные `supertest` для проверки кода состояния и заголовков, на этот раз мы проверяем данные ответа, хранящиеся в *ответе.свойство body*. Наши тесты проверяют формат и содержимое данных ответа с помощью метода strictEqual библиотеки assert.

Мы могли бы немного упростить второй тест и использовать сам assert, чтобы убедиться, что примечание находится среди возвращенных:

```
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'))
})
```

Преимущество использования синтаксиса async / await начинает становиться очевидным. Обычно нам пришлось бы использовать функции обратного вызова для доступа к данным, возвращаемым promises, но с новым синтаксисом все стало намного удобнее:

```
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)
```

Промежуточное программное обеспечение, которое выводит информацию о HTTP-запросах, препятствует выводу результатов выполнения теста. Давайте модифицируем регистратор, чтобы он не выводил данные на консоль в тестовом режиме.:

```
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
}
```

Инициализация базы данных перед тестированием

Тестирование кажется простым, и в настоящее время наши тесты проходят. Однако наши тесты плохие, поскольку они зависят от состояния базы данных, в которой теперь есть два примечания. Чтобы сделать их более надежными, мы должны перезагрузить базу данных и сгенерировать необходимые тестовые данные контролируемым образом перед запуском тестов.

В наших тестах уже используется функция after для закрытия соединения с базой данных после завершения выполнения тестов. Библиотека node: test предлагает множество других функций, которые можно использовать для выполнения операций один раз перед запуском любого теста или каждый раз перед запуском теста.

Давайте инициализируем базу данных *перед каждым тестированием* с помощью функции beforeEach:

```
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()
})
// ...
```

В начале база данных очищается, а затем мы сохраняем две заметки, хранящиеся в массиве `initialNotes`, в базе данных. Таким образом мы обеспечиваем, чтобы база данных находилась в одном и том же состоянии перед каждым тестом.

Давайте также внесём следующие изменения в последние два теста:

```
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'))
})
```

Запуск тестов один за другим

Команда `npm test` выполняет все тесты для приложения. При написании тестов обычно рекомендуется выполнять только один или два теста.

Есть несколько способов сделать это, и один из них — единственный метод. С помощью этого метода мы можем определить в коде, какие тесты должны быть выполнены:

```
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)
})
```

При запуске тестов с опцией `--test-only`, то есть с помощью команды:

{() => fs}

выполняются только `только` отмеченные тесты.

Опасность `только` в том, что можно забыть удалить их из кода.

Другой вариант — указать тесты, которые нужно запустить, в качестве аргументов команды *npm test*.

Следующая команда запускает только тесты, находящиеся в файле *tests/note_api.test.js*:

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

Параметр --tests-by-name-pattern можно использовать для запуска тестов с определенным именем:

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

Приведенный аргумент может относиться к названию теста или блоку описания. Он также может содержать только часть имени. Следующая команда запустит все тесты, в названии которых есть *примечания*:

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

Копировать

асинхронность / ожидание

Прежде чем мы напишем дополнительные тесты, давайте взглянем на ключевые слова `async` и `await`.

Синтаксис async / await, представленный в ES7, позволяет использовать *асинхронные функции, возвращающие обещание* таким образом, чтобы код выглядел синхронным.

В качестве примера выборка заметок из базы данных с помощью promises выглядит следующим образом:

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

Копировать

Метод `Note.find()` возвращает обещание, и мы можем получить доступ к результату операции, зарегистрировав функцию обратного вызова с помощью метода `then`.

Весь код, который мы хотим выполнить после завершения операции, записан в функции обратного вызова. Если бы мы захотели выполнить несколько последовательных асинхронных вызовов функций, ситуация вскоре стала бы болезненной. Асинхронные вызовы должны были бы выполняться при обратном вызове. Это, вероятно, привело бы к усложнению кода и потенциально могло бы породить так называемый ад обратного вызова.

С помощью цепочки обещаний мы могли бы держать ситуацию в некоторой степени под контролем и избежать ада обратного вызова, создав довольно чистую цепочку вызовов методов `then`. Мы видели некоторые из них во время курса. Чтобы проиллюстрировать это, вы можете просмотреть искусственный пример функции, которая извлекает все заметки, а затем удаляет первую.:

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

Копировать

Последующая цепочка в порядке, но мы можем сделать лучше. Функции генератора, представленные в ES6, предоставили умный способ написания асинхронного кода таким образом, чтобы он "выглядел синхронным". Синтаксис немного неуклюжий и широко не используется.

Ключевые слова `async` и `await`, введенные в ES7, предоставляют ту же функциональность, что и генераторы, но понятным и синтаксически более чистым способом для всех пользователей мира JavaScript.

Мы могли бы извлекать все заметки из базы данных, используя оператор await следующим образом:

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

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

Копировать

Код выглядит точно так же, как синхронный код. Выполнение кода приостанавливается на `const notes = await Note.find({})` и ожидает, пока соответствующее обещание не будет *выполнено*, а затем продолжает его выполнение до следующей строки. Когда выполнение продолжается, результат операции, вернувшей обещание, присваивается переменной `notes`.

Немного сложный пример, представленный выше, может быть реализован с помощью await, подобного этому:

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

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

Копировать

Благодаря новому синтаксису код намного проще, чем предыдущая цепочка then-chain.

Есть несколько важных деталей, на которые следует обратить внимание при использовании синтаксиса async / await. Чтобы использовать оператор await с асинхронными операциями, они должны возвращать обещание. Это не проблема как таковая, поскольку обычные асинхронные функции, использующие обратные вызовы, легко обернуть вокруг обещаний.

Ключевое слово await нельзя использовать где угодно в коде JavaScript. Использование await возможно только внутри асинхронной функции.

Это означает, что для работы предыдущих примеров в них должны использоваться асинхронные функции. Обратите внимание на первую строку в определении функции со стрелкой:

```
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()
```

В коде объявляется, что функция, назначенная `main`, является асинхронной. После этого код вызывает функцию с помощью `main()`.

асинхронность / ожидание в серверной части

Давайте начнем изменять серверную часть на async и await. Поскольку все асинхронные операции в настоящее время выполняются внутри функции, достаточно изменить функции обработчика маршрута на асинхронные функции.

Маршрут для извлечения всех заметок будет изменен на следующий:

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

Мы можем убедиться, что наш рефакторинг прошел успешно, протестировав конечную точку через браузер и запустив тесты, которые мы написали ранее.

Вы можете найти код для нашего текущего приложения полностью в *части 4-3* в этом репозитории GitHub.

Дополнительные тесты и рефакторинг серверной части

При рефакторинге кода всегда существует риск регресса, а это означает, что существующая функциональность может выйти из строя. Давайте проведем рефакторинг оставшихся операций, сначала написав тест для каждого маршрута API.

Давайте начнем с операции добавления новой заметки. Давайте напишем тест, который добавит новую заметку и проверит, увеличивается ли количество заметок, возвращаемых API, и есть ли в списке недавно добавленная заметка.

```
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'))
})
```

Тест завершается неудачей, потому что мы случайно вернули код состояния *200 OK* при создании новой заметки. Давайте изменим его на код состояния *201 СОЗДАНО*:

```
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))
})
```

Давайте также напишем тест, который проверяет, что заметка без содержимого не будет сохранена в базе данных.

```
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)
})
```

Оба теста проверяют состояние, сохраненное в базе данных после операции сохранения, извлекая все заметки приложения.

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

Те же шаги проверки будут повторяться в других тестах позже, и было бы неплохо извлечь эти шаги во вспомогательные функции. Давайте добавим функцию в новый файл с именем *tests/test_helper.js* который находится в том же каталоге, что и тестовый файл.

```
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:

```
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)
  }
})
```

Вы можете найти код для нашего текущего приложения полностью в *части 4*-ветви этого репозитория GitHub.

Устраняем ошибку try-catch

Async / await немного упрощает код, но "ценой" является структура *try / catch*, необходимая для перехвата исключений. Все обработчики маршрутов имеют одинаковую структуру

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

Возникает вопрос, возможно ли провести рефакторинг кода, чтобы исключить *catch* из методов?

В библиотеке express-async-errors есть решение для этого.

Давайте установим библиотеку

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

Использовать библиотеку *очень* просто. Вы вводите библиотеку в *app.js*, `перед` импортом маршрутов:

```
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')
  // ...
}
```

Мы сохраняем заметки, хранящиеся в массиве, в базу данных внутри цикла `forEach` . Однако, похоже, тесты не совсем работают, поэтому мы добавили несколько журналов консоли, чтобы помочь нам найти проблему.

Консоль отображает следующие выходные данные:

```
очищено
готово
введенный тест
сохранен
сохранено
```

Несмотря на использование синтаксиса async / await, наше решение работает не так, как мы ожидали. Выполнение теста начинается до инициализации базы данных!

Проблема в том, что каждая итерация цикла forEach генерирует асинхронную операцию, и `beforeEach` не будет ждать завершения их выполнения. Другими словами, команды `await`, определенные внутри цикла `forEach`, находятся не в функции `beforeEach`, а в отдельных функциях, которые не будут ждать `beforeEach`.

Поскольку выполнение тестов начинается сразу после завершения выполнения `beforeEach`, выполнение тестов начинается до инициализации состояния базы данных.

Один из способов исправить это - дождаться завершения выполнения всех асинхронных операций с помощью метода Promise.all:

```
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)
})
```

Решение довольно продвинутое, несмотря на свой компактный внешний вид. Переменная `noteObjects` присваивается массиву объектов Mongoose, которые создаются с помощью конструктора `Note` для каждой из заметок в массиве `helper.initialNotes`. Следующая строка кода создает новый массив, *состоящий из promises*, которые создаются путем вызова метода `save` для каждого элемента в массиве `noteObjects`. Другими словами, это набор обещаний для сохранения каждого из элементов в базе данных.

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*.

Клятва настоящего разработчика Full stack

Создание тестов — ещё один уровень сложности в программировании. Мы должны обновить нашу клятву разработчика полного цикла, чтобы напомнить вам, что при разработке тестов также важна систематичность.

Итак, мы должны еще раз продлить нашу клятву:

Разработка полного стека *очень сложна*, поэтому я буду использовать все возможные средства, чтобы упростить её

- У меня все время будет открыта консоль разработчика моего браузера
- Я воспользуюсь вкладкой «Сеть» в инструментах разработчика браузера, чтобы убедиться, что интерфейс и сервер взаимодействуют так, как я ожидаю
- Я буду постоянно следить за состоянием сервера, чтобы убедиться, что данные, отправленные на него с помощью интерфейса, сохраняются так, как я ожидаю
- Я буду следить за базой данных: сохраняет ли серверная часть данные в правильном формате
- Я буду продвигаться небольшими шагами
- *Я напишу много операторов `console.log`, чтобы убедиться, что я понимаю, как работает код и тесты, и чтобы выявить проблемы*

- Если мой код не работает, я не буду писать новый. Вместо этого я начинаю удалять код, пока он не заработает, или просто возвращаюсь к состоянию, когда всё ещё работало
- *Если тест не проходит, я удостоверяюсь, что тестируемый функционал точно работает в приложении*
- Когда я обращаюсь за помощью на канале course Discord или где-либо еще, я правильно формулирую свои вопросы, смотрите Здесь, как обратиться за помощью

Упражнения 4.8.-4.12.

Предупреждение: Если вы обнаружите, что используете методы async / await и *then* в одном и том же коде, почти гарантированно, что вы делаете что-то неправильно. Используйте один или другой вариант и не смешивайте их.

4.8: Тесты списка блогов, шаг 1

Используйте библиотеку SuperTest для написания теста, который отправляет HTTP GET-запрос на */api/blogs* URL. Убедитесь, что приложение blog list возвращает правильное количество записей в блоге в формате JSON.

После завершения теста реорганизуйте обработчик маршрута, чтобы использовать синтаксис async / await вместо promises.

Обратите внимание, что вам придется внести в код изменения, аналогичные тем, которые были внесены в материал, например, определить среду тестирования, чтобы вы могли писать тесты, использующие отдельные базы данных.

ПРИМЕЧАНИЕ: когда вы пишете свои тесты, ***лучше не выполнять их все***, выполняйте только те, над которыми вы работаете. Подробнее об этом здесь.

4.9: Тесты списка блогов, шаг 2

Напишите тест, который проверяет, что свойство уникального идентификатора записей блога называется *id*, по умолчанию база данных называет свойство *_id*.

Внесите необходимые изменения в код, чтобы он прошел тест. Метод toJSON, описанный в части 3, является подходящим местом для определения параметра *id*.

4.10: Тесты списка блогов, шаг 3

Напишите тест, который проверяет, что выполнение HTTP-запроса POST к URL-адресу */ api / blogs* успешно создает новую запись в блоге. По крайней мере, убедитесь, что общее количество блогов в системе увеличилось на один. Вы также можете убедиться, что содержимое записи в блоге правильно сохранено в базе данных.

Как только тест будет завершен, реорганизуйте операцию, чтобы использовать async / await вместо promises.

4.11 *: Тесты списка блогов, шаг 4

Напишите тест, который проверяет, что если свойство *likes* отсутствует в запросе, то по умолчанию оно будет равно 0. Пока не тестируйте другие свойства созданных блогов.

Внесите необходимые изменения в код, чтобы он прошел тест.

4.12 *: Тесты списка блогов, шаг 5

Напишите тесты, связанные с созданием новых блогов через конечную точку */api/blogs*, которые проверяют, что если в данных запроса отсутствуют свойства *title* или *url*, серверная часть отвечает на запрос кодом состояния *400 Bad Request*.

Внесите необходимые изменения в код, чтобы он прошел тест.

Тесты рефакторинга

В настоящее время наше тестовое покрытие отсутствует. Некоторые запросы, такие как *GET /api/notes /:id* и *DELETE /api/notes /: id*, не тестируются, когда запрос отправляется с недопустимым идентификатором. Группирование и организация тестов также нуждаются в некотором улучшении, поскольку все тесты существуют на одном и том же "верхнем уровне" в тестовом файле. Читаемость теста улучшится, если мы сгруппируем связанные тесты с помощью блоков *describe*.

Ниже приведен пример тестового файла после внесения некоторых незначительных улучшений:

Копировать

```
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()
})
```

Выходные данные теста в консоли сгруппированы в соответствии с блоками *описания*:

```
▶ 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