Estructura de la aplicación backend, introducción a las pruebas

Estructura del proyecto

Como quedaría nuestro proyecto luego de la reestructuración

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

Ahí quedaría

Otra modificación seria que dejemos de usar **console.log** o **console.error** para mostrar los errores y empezar a usar un funciones que lo hagan y que tengan sus propios modulos por ejemplo una función para los errores o para información en general por ejemplo se veria asi

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


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

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

Esto sería el documento **logger.** Tiene beneficios hacer esto porque si en un futro queremos realizar cambios solo lo tenemos que hacer en un lugar y como quedaría el archivo **index** de nuestro backend despues del reordenamiento

```
const app = require('./app') // la aplicación Express real
const http = require('http')
const config = require('./utils/config')
const logger = require('./utils/logger')


const server = http.createServer(app)


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

Ahora todo nuestro backend estaría funcionando en el archivo **app.js(no confundir con el archivo de react)** e **index.js** se encargaría de importarlo e iniciarlo

Las **variables de entorno** ahora tienen su propio archivo que es **config.js** dentro de la carpeta **utils** las funciones de ayuda como **info** están dentro de **logger** que a su vez también esta en la carpeta **utils**

Como accederíamos a las **variables de entorno** desde otro archivo? Así

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

Los **controladores de rutas** tambien tienen su propio modulo **controllers**y todas las rutas ahora están ahí

Vemos como esta quedando:

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

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

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

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

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

  note.save()
    .then(savedNote => {
```

```
      response.json(savedNote)
    })
    .catch(error => next(error))
})

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

Nos dimos cuenta de la diferencia que es la palabra **notesRouter** que ahora aparece que es un objeto **rounter**

Ahora todas las rutas están dentro de este objeto también notamos que las rutas ahora están acortadas antes se veian asi:

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

y ahora se ven asi

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

Este **enrutador** lo tenemos que llevar al archivo **app.js** que se veria asi

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

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

En esta ultima linea vemos como definimos una ruta esto quiere decir.

El enrutador se usa si la **url** empieza en(la ruta definida en cuestion)

Como quedaría **app.js** con todo esto?

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

const express = require('express')

const app = express()

const cors = require('cors')

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

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

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

const mongoose = require('mongoose')


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


mongoose.connect(config.MONGODB_URI)
```

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

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

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

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

module.exports = app}
```

Ahora veamos como queda nuestro archivo donde esta el middleware

```
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 === 'ValidatioEnError') {
    return response.status(400).json({ error: error.message })
  }

  next(error)
}

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

Establecer conexión con la base de datos

Esa responsabilidad se la hemos delegado al archivo **app.js**. El archivo **note.js** del directorio **models** solo define el esquema para las notas

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

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

```
  date: {
    type: Date,
    required: true,
  },
  important: Boolean,
})

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

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

Testing de aplicaciones Node

Esto es Nuevo y son las pruebas automatizadas. Cuando el proyecto es tan simple como el que estamos haciendo no tiene sentido hacer pruebas unitarias

Pero vamos a poner un ejemplo

Para empezar vamos a crear un nuevo archivo dentro de la carpeta **utils** que se llame **for__testing.js** y haces que se vea asi

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

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

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

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

Para los test nosotros vamos a usar la librería **jest** vamos a instalarla

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

dentro de **package.json** creamos un script para iniciar los tests

```
{
  //...
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
     "build:ui": "rm -rf build && cd ../../../2/luento/notes && npm run build && cp -r build ../../../3/luento/notes-backend",
    "deploy": "git push heroku master",
     "deploy:full": "npm run build:ui && git add . && git commit -m uibuild && git push && npm run deploy",
    "logs:prod": "heroku logs --tail",
    "lint": "eslint .",

    "test": "jest --verbose"
  },
  //...
}
```

Jest requiera que se le indique el entorno de ejecución esto lo hacemos en el **package.json** agregando al final esto

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

Jest también podría buscar un archivo de configuración con el nombre predeterminado **jest.config.js** que se veria asi

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

Creamos un archivo separado para el test

Este archivo estará en su propia carpeta que llamaremos **tests** que va a tener un archivo llamado **palindrome.test.js** que tendrá dentro el siguiente código

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


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


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


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


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

```
test('palindrome of releveler', () => {
  const result = palindrome('releveler')


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

En el caso de que tengamos instalado **ESlint** nos va a dar un monton de errores por las palabras clave **test** y **expect**

Anulemos esto agregando **"jest":true** a la propiedad **env** del archivo **.eslintrc.js.**

Se veria asi

```
module.exports = {
  "env": {
    "commonjs": true
    "es6": true,
    "node": true,


    "jest": true,
  },
  "extends": "eslint:recommended",
  "rules": {
    // ...
  },
};
```

Analicemos el código del archivo del test

La primera línea es:

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

donde importamos el archivo sobre el cual aplicamos el **test**

```
test('palindrome of a', () => {
```

```
  const result = palindrome('ab')


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

Analicemos esto recibe dos parámetros el primero es la descripción y el segundo la función que define la funcionalidad para casa prueba que es este caso espera el texto al revés

Probando el backend

Ahora mismo no tiene mucho sentido hacer pruebas unitarias en nuestro proyecto por lo sencillo que es.

Ahora bien a veces es mejor hacer las pruebas del backend sobre una base de datos simulada en vez de la original.

Para esto se puede usar la biblioteca **mongo-mock**

En esta sección vamos a probar la aplicación a través de su **API REST** y que de paso la a base de datos este también incluida

Entorno de prueba

Vamos a configurar las cosas para tener un servidor para el mode de desarrollo y otro para el modo producción

En las pruebas es una buena practica hacer esto. Y obviamente esto lo configuramos desde el archivo package.json que quedaría asi

```
{
  // ...
  "scripts": {

    "start": "NODE_ENV=production node index.js",
    "dev": "NODE_ENV=development nodemon index.js",
    "build:ui": "rm -rf build && cd ../../../2/luento/notes && npm run build && cp -r build ../../../3/luento/notes-backend",
    "deploy": "git push heroku master",
    "deploy:full": "npm run build:ui && git add . && git commit -m uibuild && git push && npm run deploy",
    "logs:prod": "heroku logs --tail",
    "lint": "eslint .",

    "test": "NODE_ENV=test jest --verbose --runInBand"
  },
  // ...
}
```

Tambien agregamos dentro de test **runInBand** para que **jest** no ejecute pruebas en paralelo

Todo esto que hacemos esta bien pero en Windows **no funciona** para arreglar esto vamos a necesitar de una librería que nos ayude que es **cross-env** que lo instalamos asi

npm install --save-dev cross-env

y ahora si logramos una compatibilidad multiplataforma

Hagamos unos cambios dentro del archivo **config.js** que quedaría asi

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

const PORT = process.env.PORT


let MONGODB_URI = process.env.MONGODB_URI

if (process.env.NODE_ENV === 'test') {

  MONGODB_URI = process.env.TEST_MONGODB_URI

}


module.exports = {

  MONGODB_URI,

  PORT

}
```

El archivo **.env** tiene la variables independientes para las direcciones **url** que son para la base de datos de prueba y la de desarrollo

Que se ve asi

```
MONGODB_URI=mongodb+srv://fullstack:secred@cluster0-ostce.mongodb.net/note-app?retryWrites=true

PORT=3001


TEST_MONGODB_URI=mongodb+srv://fullstack:secret@cluster0-ostce.mongodb.net/note-app-test?retryWrites=true
```

Supertest

Es un paquete que va a ayudarnos a probar nuestra **API**

La instalamos asi:

npm install --save-dev supertest

Ahora escribamos la primera prueba dentro del archivo **tests/note_api.test.js**

Que se veria asi

```
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(() => {

  mongoose.connection.close()

})
```

Analicemos el test primero el test importa **app** que recordemos es la aplicación

Que luego se envuelve con la función **supertest** esto internamente la guarda en un objeto llamado **superagent** este objeto se asigna a la variable **api**

Quedando asi `const api = supertest(app)`

La prueba continua haciendo una solicitud **get** a una **url** que se verifica si se devuelve un código de estado **200** tambien se espera que el encabezado tenga un formato **json**

Una vez terminadas las prueba cerramos la conexión con la función **AfterAll**

A veces la versión 6 de **mongo** nos da un problema en este punto

Podemos arreglar esto creando dentro de la carpeta **tests** el archivo **teardown.js** con el siguiente contenido

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

Dentro del archivo **package.json** en la definición de **jest** lo dejamos asi

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

   "globalTeardown": "./tests/teardown.js"
 }
}
```

Otro error que puede ocurrir es que **jest** espere menos tiempo de lo que necesita el servidor para devolver los datos ahí podemos agregar mas tiempo se vería así:

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

Al final vemos como agregamos un numero ese es el tiempo a esperar

Vamos a usar **supertest** dentro del archivo **app.js** que se veria asi

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

```
// ...
```

Escribiendo mas pruebas

```
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')
})
```

La explicación de los test es bastante obvia solo hay que leerla

Teniamos dos funciones **middleware** que se encargaban de los mensajes por consola editemosla para que solo se ejecuten en modo desarrollo lo hacemos asi

Inicializando la base de datos antes de las pruebas

Por el momentos las pruebas están siendo simples pero no son buenas ya que dependen del estado de la base de datos.

Para hacer las pruebas mas robustas es necesario restablecer la base de datos y generar los datos de prueba de manera controlada

Por ahora estuvimos usando **afterAll** que es una función que se ejecuta al final siempre pero hay otras funciones permitidas en **Jest** por ejemplo una es **beforeEach** que se encarga en nuestro caso de iniciar nuestra base de datos antes de cada prueba. Se veria asi

```
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',
    date: new Date(),
    important: false,
  },
  {
    content: 'Browser can execute only Javascript',
    date: new Date(),
    important: true,
  },
]

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

// ...
```

Que vemos que pasa? Dentro de **beforeEach** indicamos que al principio borramos toda la base de datos y luego agregamos dos notas que están dentro del array **initialNotes** con esto nos aseguramos de que siempre al iniciar la base este en el mismo estado.

Agreguemos unos cambios antes

```
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'
  )
})
```

toContain se encarga de verificar que por lo menos una nota compla con lo que se pasa como parámetros

Ejecución de pruebas una por una

El comando **npm test** se encarga de ejecutar todas las pruebas de la aplicación. Es recomendable no mas de 1 o 2 podemos ejecutar pruebas en especifico agregándolo como parámetro en la terminal asi

npm test -- -t 'a specific note is within the returned notes'

Ahi solo ejecutariamos esa prueba

Async-Await

Esto hace posible que usemos funciones asíncronas que devuelvan promesas y que parezca sincronico

Un ejemplo de como se obtienen notas de una base de datos con promesas es la siguiente

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

Note.find devuelve una promesa y **then** nos permite ingresar dentro.

Pero andar usando varias llamadas se empezaría a complicar todo y encadenar promesas como hicimos en capítulos anteriores no arregla todo

Con **async-await** la cosa funcionaria asi

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

Mucho mas simple todo y mas resumido

La aplicación se detiene en **const notes = awit.find({})** y le asigna el resultado a **notes** y hasta que no se cumpla no continua con la siguiente línea

Aca vemos otro ejemplo

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

const response = await notes[0].remove()

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

Uno no puede usar **await** asi nomas tiene que ser dentro de una function **async** como el siguiente ejemplo

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


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

main()
```

ahí si se puede usar **await**

Async-await en el backend

Modifiquemos todo el backend empezando por las rutas para que utilice **async-await**

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

Ahora ya no vamos a tener problemas con que la **api** tarde en responder y no de error eso

Mas pruebas y refactorización del backend

Cuando se refactoriza existe siempre el riesgo de **regresión** que significa que código que funcionaba deje de hacerlo

En este caso vamos a escribir una prueba para cada ruta

Comencemos con la operación para agregar una nota nueva

Y que la cantidad de notas devueltas por la **api** aumente en una por esta ultima que agregamos. Se veria asi

```
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(200)
    .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'
  )
})
```

Esta prueba hace todo lo que describimos y funciona correctamente ahora hagamos una cosa más vamos a verificar que no se guarden notan sin contenido.

```
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)
})
```

En ambos ejemplos la promesa se guarda dentro de **response** y es **response** quien sometemos a las pruebas

Estos mismos pasos de verificación los realizaremos mas adelante por eso es buena idea **extraer** estos pasos en funciones auxiliares.

Vamos a agregar la función a un nuevo archivo llamado **test/test_helper.js**

Que va a estar en el mismo directorio que el archivo de prueba

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

const initialNotes = [
  {
    content: 'HTML is easy',
    date: new Date(),
    important: false
  },
  {
    content: 'Browser can execute only Javascript',
```

```
    date: new Date(),
    important: true
  }
]

const nonExistingId = async () => {
  const note = new Note({ content: 'willremovethissoon', date: new
Date() })
  await note.save()
  await note.remove()

  return note._id.toString()
}

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

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

El módulo define la función **notesInDb** que se puede usar para verificar las notas almacenadas en la base de datos. La matriz **initialNotes** que contiene el estado inicial de la base de datos también está en el módulo. También definimos la función **nonExistingId** con anticipación, que se puede usar para crear un ID de objeto de base de datos que no pertenezca a ningún objeto de nota en la base de datos.

Ahora podemos usar esta función auxiliar importándola desde el archivo donde están nuestros **test** asi quedaría

```
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(200)
    .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(() => {
  mongoose.connection.close()
})
```

Hora de refactorizar el código con Async-Await

Asi quedaría una de las rutas con **asyn await**

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

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

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

Pero hay un problema le falta una forma de manejar los errores eso lo vamos a gestionar con **catch**

Manejo de errores y Asyn-Await

Que pasa si hay un error y no gestionamos eso. Bueno quela solicitud nunca va a recibir una respuesta. Esto lo vamos a arreglar con **try/catch**

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

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

  try {
    const savedNote = await note.save()
    response.json(savedNote)
  } catch(exception) {
    next(exception)
  }
})
```

Catch de lo que se encarga es de enviar el **error** a el **middleware** de gestión de errores. Debemos hacer lo mismo con todas las rutas.

```
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)
  }
})
```

Eliminando el try-catch

Si alguien es tan sensible de que le moleste una estructura básica del lenguaje como lo es **try/catch** existe una librería que permite no tener que usarlo, se llama **express-async-error** pero solo lo nombro usarlo me parece de trolo

Optimizacion de la función beforeEach

Volvamos al archivo donde están nuestros **tests** anteriormente teníamos la función **beforeEach** y se veía asi

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

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

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

Pero hay una mejor forma y es esta

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

Esto nos puede dar unos errores porque no todas las promesas aun fueron cargadas esto lo podemos arreglar con **promise.all** que hace que espere que todas las promesas se terminen de ejecutar primero quedando el código asi

```
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)
})
```

Releer esta parte de la teoria

Administración de usuarios

Queremos agregar autenticación y autorización de usuario a nuestra aplicación. Los usuarios deben almacenarse en la base de datos y cada nota debe estar vinculada al usuario que la creó

.Comenzamos agregando información de los usuarios a la base de datos.

Debe existir una relación de 1 a 1 entre **Note** y **User**

En **MongoDB** no hay una manera única de relacionar las notas y los usuarios si queremos manetener las estructuras que manejábamos anteriormente podemos crear una colección para los usuarios a la que vamos a llamar **users**

Vamos a utilizar el **id** para hacer referencia a documentos de otras colecciones esto en otro tipo de bases de datos lo logramos con **consultas de unión** cosa que en **mongo** no podemos hacer pero que a partir de la versión **3.2** de **mongo** existen las **consultas de agregación de búsqueda** aunque no la examinaremos en este curso.

Referencia entre colecciones

Podemos hacer que cada **nota** tenga como referencia al **usuario** que la creo.

Supongamos una colección de **users** que contiene dos usuarios.

```
[
	{
		username: “mluukai”,
		_id: 123456,
	},
	{
		username: “hellas”,
		_id: 141414,
	}
]
```

Ahora vayamos a la colección **notes** y veamos que hay 3 notas que tienen un campo user que hace referencia a un usuario en la colección **users**

Escribamos uno como ejemplo

```
[
	{
		content: “soy el ejemplo de una nota”,
		important: false,
		_id: 221212,
		user: 123456
	},
]
```

No es estrictamente necesario que dentro de cada nota este el **id** del que lo creo también podríamos dentro de la colección de **usuarios** que cada usuario tenga un **array** con el **id** de cada **nota** creada asi

```
[
	{
		username: “mluukai”,
		_id: 123456,
		Notes:[221212,1231231]
	},
]
```

Las **bases de datos de documentos** ofrecen la posibilidad de alojar todo en una misma nota se veria asi

```
[
	{
		username: “mluukai”,
		_id: 123456,
		Notes:[
			{
				content: “soy el ejemplo de una nota”,
```

```
                        important: false,
                        _id: 221212,
                        user: 123456
                    },
                ]
        ]
        },
]
```

Así quedaría de esta forma pero en este tipo de base de datos la decisión depende del diseño más conveniente para el sistema.

Esquema de Mongoose para usuarios

En este caso tomamos la decisión de almacenar el id de las notas dentro del archivo del usuario

Quedaría así.

```
const mongoose = require("mongoose")
const userSchema = new mongoose.Schema({
        username: String,
        name: String,
        passwordHash:String,
        notes: [
                {
                        type: mongoose.schema.Type.ObjectId,
                        ref: "Note"
                }
        ]
})
userSchema.set("toJSON",{
        transform: (document, returnedObject) => {
                returnedObject.id = returnedObject._id.toString()
                delete returnedObject._id
                delete returnedObject.__v
```

```
            delete  returnedObject.passwordHash
      }
})
const User = mongoose.model("User", userSchema)
module.exports = User
```

los **indentificadores** de **notas** se almacenan dentro del documento del **usuario** como un array de objetos **ID** de **mongo** esta es la definición

```
{
      Type: mongoose.Schema.Type.ObjectId,
      Ref: "Note"
}
```

El tipo de campo es **ObjectId** que hace referencia a documentos de estilo-**nota**

Mongo no sabe que este campo hace referencia a **notes** la sintaxis esta relacionada y definida por **mongoose**

Expandamos el esquema de la nota definida en el archivo **model/note.js**

```
const noteSchema = new mongoose.Schema({
      content: {
            type: String,
            required: true,
            minlength: 5
      },
      Date: Date,
      Important: Boolean,
      User: {
            type: mongoose.Schema.Type.ObjectId,
            ref: "User"
      }
})
```

La nota hace referencia al usuario que la creo y los usuarios tienen las referencias a las notas creadas por si mismos

Creando usuarios

Creemos una nueva ruta para crear usuarios los usuarios tienen **nombre de usuario** único, un nombre y algo que llamamos **contraseñaHash**

El **hash** de la contraseña es el resultado de una función **unidireccional** esto lo hacemos asi por que no es seguro guardar las contraseñas como **texto plano**

Para generar los **hashes** de las contraseñas vamos a usar un paquete llamado **bcrypt** para instalarlo usamos el comando

La creación de usuarios lo hacemos siguiendo las convenciones de **RESTful**

Empecemos

Vamos a definir un **enrutador** separado para tratar con los usuarios. Crendo un archivo **users.js** dentro del directorio **controllers**

luego usaremos ese **enrutador** en la aplicacion en el archivo **app.js** de modo que maneje las solicitudes a **/api/users** quedando asi.

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

// ...

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

el contenido del archivo que define el enrutador es este

```
const bcrypt = require('bcrypt')

const usersRouter = require('express').Router()

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


usersRouter.post('/', async (request, response) => {

  const body = request.body


  const saltRounds = 10
```

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

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

  const savedUser = await user.save()

  response.json(savedUser)
})

module.exports = usersRouter
```

Vemos que la contraseña que nosotros ingresamos no se guarda en la base de datos lo que se guarda en la base de datos es el **hash** de la contraseña que se genera con la funcion **bcrypt.hash**

Esto luego lo tenemos que probar pero es bastante engorroso poner a hacerlo manualmente en **postman** podriamos usar **pruebas automatizadas** que se veria asi

```
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(200)
      .expect('Content-Type', /application\/json/)

    const usersAtEnd = await helper.usersInDb()
    expect(usersAtEnd).toHaveLength(usersAtStart.length + 1)

    const usernames = usersAtEnd.map(u => u.username)
    expect(usernames).toContain(newUser.username)
  })
})
```

Las pruebas usan la funcion auxiliar **usersInDb()** que la habiamos definido en el archivo **test_helper.js** de la carpeta **test** que ayuda a verificar el estado de la base de datos despues de que se crea un usuario

El bloque **beforeEach** agrega un usuario con el nombre de usuario root a la base de datos. La función se utiliza para ayudarnos a verificar el estado de la base de datos después de que se crea un usuario:

Este era la funcion

```
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,
}
```

Ahora veamos esto

```
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/)

    expect(result.body.error).toContain('`username` to be unique')

    const usersAtEnd = await helper.usersInDb()
    expect(usersAtEnd).toHaveLength(usersAtStart.length)
  })
})
```

El caso de prueba obviamente no pasará en este punto. Básicamente, estamos practicando **desarrollo impulsado por pruebas (TDD)**, donde las pruebas para la nueva funcionalidad se escriben antes de implementar la funcionalidad.

Singularidad del nombre

para validar que un campo sea unico existe un paquete llamado **mongoose-unique-validator** que lo instalamos asi

npm install mongoose-unique-validator

Seguidos de cambios en el **esquema** de la ruta **models/users.js**

```
const mongoose = require('mongoose')
const uniqueValidator = require('mongoose-unique-validator')
const userSchema = new mongoose.Schema({
  username: {
    type: String,
```

```
    unique: true
  },
  name: String,
  passwordHash: String,
  notes: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Note'
    }
  ],
})

userSchema.plugin(uniqueValidator)
```

Se podrian comprobar mas cosas como que el largo de un campo entre otras cosas.

Agreguemos un implementador d ruta que devuelve todos los usuarios de la base de datos

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

Creacion de una nueva nota

El codigo que crear una nueva nota debe actualizarse para que la nota se asigne al usuario que la creo. Vamos a modificar lo que tenemos para que **la informacion del usuario** se **envie** a el campo **userId** del cuerpo de la solicitud

```
const User = require('../models/user')
//...
notesRouter.post('/', async (request, response, next) => {
  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,
    date: new Date(),

    user: user._id
  })

  const savedNote = await note.save()

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

  response.json(savedNote)
})
```

Analicemos el codigo

primer definimos el tipico **body** luego definimos **user** que es el usuario que tiene la misma **id** que el usuario que creo la nota es decir buscamos en nuestra lista de usuarios al usuario con el **id** correspondiente. luego vemos la creacion del objeto **note** que se forma de los elemento pasados como artibutos dentro de **body** y tambien con el **id** que se encuetra dentro del **user** que definimos antes luego definimos **savedNote** que es la nota.

tambien al **user** que definimos al comienzo que es el usuario en si le agregamos al atributo **notes,** que es un array con todas las notas guardadas, el **id** de la nota que creamos ahora.

Poblar

Estamos buscando que cuando vayamos a la ruta **/api/users** cada objeto usuario tenga a su vez las notas que el usuario creo y no

solamente sus identificadores en **MySQL** eso lo lograriamos con **una consulta de combinacion** pero no existe en mongo.

Pero **mongoose** puede ayudar a hacer algunas uniones con consultas en segundo plano usando el metodo **populate**

actualizemos el codigo para que se vea asi.

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

El metodo **populate** se encadena atras de **find** para decirle a **mongoose** que cargue las notas a las que se hace referencia en **notes** que suponemos es un array de **ids** es decir buscamos las notas que tengan esos **ids** y **populate** se encargara de cambiar los **ids** por las notas con esos **ids**

Para especificar los campos especificos que debe incluir la respuesta agregamos un objeto tambien como paramentro de **populate** asi quedaria

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

Recordemos que la base de datos no saque los que los **id** en **user** hacen referencia a documentos en la coleccion **usuarios** pero si recordemos que agregamos referencias

```
const noteSchema = new mongoose.Schema({
  content: {
    type: String,
    required: true,
    minlength: 5
```

```
  },
  date: Date,
  important: Boolean,
  user: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  }
})
```

Autenticacion de token

Los usuarios deben poder **iniciar sesion** en la aplicacion cuando uun usuario lo haga su informacion debe adjuntarse automaticamente en cualquier **nota** que cree.

Ahora implementemos un soporte para **autenticacion basada en token** para el backend, veamos la frecuencia en la que funciona esto.

- El usuario inicia sesion usando un formulario implementado en react
- **React** envia los el **usuario** y la **contraseña** al servidor a una direccion **/api/login** con **HTTP POST.**
- Si el nombre de usuario y la contraseña son correctos se crea un **token** que identifica al usuario que inicio sesion(token firmado digitamente)
- El backend responde con un codigo **200**
- El navegador guarda el **token** por ejemplo en el estado de la aplicacion
- Cuando el usuario crea una nueva nota el codigo react envia un **token** al servidor con la solicitud.
- El servidor usa al **token** para identificar al usuario.

Primero instalemos **jsonwebtoken** que nos permite crear **tokens web JSON** Lo hacemos asi.

npm install jsonwebtoken

El codigo para la funcionalidad de inicio de sesion va a los controladores de **controllers/login.js** que se ven asi.

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

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

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

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

module.exports = loginRouter
```

Analicemos el codigo.
Primero vemos que buscamos en la base de datos al usuario por el nombre de usuario adjunto a la solicitud. Luego verificamos la contraseña que tambien se adjunta a la solicitud. Recordemos que la contraseña no se guarda en la base de datos directamente sino el **hash** usamos el metodo **bcrypt.compare** para verificar el **hash** de la contraseña ingresada con el **hash** correcto guardado en la base de datos.
en caso de que no sea correcta la verificacion devolvemos un error de tipos **401 unauthorized.**
Si la contraseña es la correcta se crea el **token** con el metodo **jwt.sign.**
Este token contiene el nombre de usuario y la identidicacion del usuario en el formulario firmado digitalmente.
Asi se veria el **token.**

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

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

El **token** creado esta firmado digitalmente usando una variable de entorno **SECRET** como secreto. La firma digital garantiza que solo las partes que conocen el secreto puedan general un token valido. El valor de la variable de entorno debe establecerse en el archivo **.env**
Una **solicitud exitosa** se **responde** con el codigo 200
Ahora el codigo de inicio de sesion debe agregarse a la aplicacion agregando un el nuevo enrutador a **app.js**

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

//...

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

Limitación de la creación de nuevas notas a los usuarios registrados

modifiquemos la creacion de notas para que solo sea posible si la nota tiene adjuntado un **token** valido.
Hay varias formas de enviar el **token** del navegador al servidor en este caso usaremos el encabezado **Authorization.** El encabezado tambien indica el **esquema de autenticacion** esto es util si es que se ofrecen varias formas de autenticacion. La identificacion del esquema le dice al servidor como interpretar las credenciales adjuntas.
El **esquema Bearer** se adapta a nuestras necesidades.
Esto que significa? Que si el valor del **token** es una cadena tal que asi

```
eyJhbGciOiJIUzI1NiIsInR5c2VybmFtZSI6Im1sdXVra2FpIiwiaW
```

El encabezado de autorizacion tendra un valor de

```
Bearer eyJhbGciOiJIUzI1NiIsInR5c2VybmFtZSI6Im1sdXVra2FpIiwiaW
```

Ahora la creacion de notas se vera asi.

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

// ...

const getTokenFrom = request => {
  const authorization = request.get('authorization')
  if (authorization && authorization.toLowerCase().startsWith('bearer ')) {
    return authorization.substring(7)
  }
  return null
}

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

  const token = getTokenFrom(request)
  const decodedToken = jwt.verify(token, process.env.SECRET)
  if (!token || !decodedToken.id) {
    return response.status(401).json({ error: 'token missing or invalid' })
  }
  const user = await User.findById(decodedToken.id)

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

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

  response.json(savedNote)
})
```

Analicemos el codigo.
Vemos la a la funcion **getTokenFrom** que lo que hace es aislar el encabezado **authorization.** La validez del **token** se comprueba con **jwt.verify.** Que tambien decodifica el token y devuelve el objeto en el que se baso el **token**

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

La verificacion del **token** tambien puede causar un error **JsonWebTokenError** si es que es invalido o esta ausente. Extendamos nuestro **middleware** para tener en cuenta este caso en particular.

```
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 ===  'JsonWebTokenError') {
    return response.status(400).json({ error: error.message })
  }

  next(error)
}
```

El **token** tiene dos campos **username** y **id** que le dice al servidor quien hizo la conexion.
Si no hay **token** o el objeto decodificado del token no contine la identidad del usuario(**decodedToken.id** no esta definido) El codigo devuelve un estado de error **401 unauthorized** y eso explica esta respuesta.

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

Cuando al fin resolvemos la identidad del autor de la solicitud la ejecucion continua como antes.
Ahora se puede crear una nueva nota de usuario usando **postman**
si el encabezado **authorization** tiene el valor correcto la cadena **bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ**
donde el segundo valor es el **token** devuelto por la operacion **inciar sesion**.
nota! Si la aplicacion tiene multiples interfaces que requieren identificacion la validacion de JWT debe separarse en su propio middleware o se pueden usar librerias como express-jwt

Problemas de la autenticacion basada en Tokens

La autenticacion basada en token es facil de implementar pero tienen un problema y es que una vez que el cliente de la **API** tiene el **token** la confianza de la **API** es **ciega.**
Supongamos que queremos revocar los derechos de acceso del titular del **token** hay dos formas de hacer eso.
una limitando el periodo de validez del 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 })
})
```

Y cada vez que el usuario necesita un **token** nuevo por que el anterior caduco debe iniciar sesion de nuevo.
El **middleware** de manejo de errores debe expandirse para dar un error adecuado al caso de un **token** caducado.

```
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 === 'JsonWebTokenError') {
    return response.status(401).json({
      error: 'invalid token'
    })

  } else if (error.name === 'TokenExpiredError') {
    return response.status(401).json({
      error: 'token expired'
    })
  }

  next(error)
}
```

Mientras mas corto sea el tiempo de caducidad mas segura sera la solucion. Por lo tanto si el **token** cae en manos equivocadas no hace falta revocar el acceso del usuario al sistema. Pero esto genera un dolor para el usuario y una molestia.
Otra solucion es guardar informacion sobre cada **token** en la base d edatos y verificar en cada solicitud de **API** si el derecho de acceso correspondiente al **token** sigue siendo valido. Con este esquema los derecjos de acceso pueden ser revocados en cualquier momento a esto se lo llama **server-side session**
Pero es mas complejo en backend esto. y menor rendimiento. por que se verifica muy seguido. Es por eso que es comun guardar la sesion correspondiente a un **token** en una **base de datos de llave-valor** como lo es **redis.**
Es comun que en vez de encabezados se usen **cookies.**