**CURSO PROFESIONAL DE NODE.js y JS**

**Backend for frontend**: Tecnica de node.js para el backend, el frontend es donde recibimos todas nuestras peticiones HTTP del cliente y en el backend for frontend realizamos conexión a base de datos y demás cosas puramente del backend.

Como crear una cuenta de Amazon Web Services y configurar S3(Almacenamientos de archivos)

La plataforma donde haremos deploy de nuestra aplicación será Amazon Web Services, la cual, incluye muchos servicios que van desde infraestructura de servidores, almacenamiento de datos o archivos (S3), dashboards, etc. Crear una cuenta en Amazon implica usar de una tarjeta de crédito, pero no te preocupes, no te genera cobro si no excedes los límites de prueba.

Configurar S3 es muy sencillo, tenemos la opción de crear un nuevo bucket, lo cual necesita:

\*El nombre del Bucket  
 \*La región (entre que el centro de datos esté más cerca, es menos la latencia)

En S3 podemos agregar carpetas o subir archivos desde la interfaz ewb, pero el mejor valor lo obtienes al usar el API y subir esos archivos desde su aplicación.

**No vamos a tener cuenta en S3 porque no tengo Tarjeta de crédito☹, igual voy a explicar cómo sería todo el proceso de este y lo tendré todo anotado.**

Usando el SDK de Amazon Web Services

Vamos a instalar el multer-s3 y el AWS de amazon.

npm install aws-sdk --save

npm install multer-s3 –save

Para trabajar con AWS debemos tener las credenciales de seguridad, es una buena práctica poner estas palabras claves o credenciales de seguridad en variables de entorno del sistema operativo, para si no incluir estas dentro de nuestro código y así no hacemos una subida de estas a repositorios o nuestro almacenamiento en la nube.

Las variables de entorno están creadas dentro de la sesión de ejecución de esta aplicación, son variables que están disponibles para las aplicaciones que están haciendo uso de esta sesión.

Vamos a crear un archivo config.js, y dentro de estos vamos a crear un objeto que obtenga nuestras llaves públicas y privadas que estarán en variables de entorno:

'use strict'

const config = {

aws: {

accessKey: process.env.AWS\_ACCESS\_KEY,

secretKey: process.env.AWS\_SECRET\_KEY

}

}

module.exports = config;

**Documentación de multer-S3:**

<https://www.npmjs.com/package/multer-s3>

**Documentación de AWS-SDK:**

<http://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/getting-started-nodejs.html>

Si queremos saber más a fondo sobre la configuración del SDK de aws mirar la documentación que esta arriba.

Ahora nos posamos en nuestro archivo de server.js donde vamos a requerir nuestros modulos del aws-sdk, el archivo donde obtenemos nuestras keys de AWS y el modulo del código que nos da multer para manejar el S3

var config = require('./config');

var aws = require('aws-sdk');

var multerS3 = require('multer-s3');

Luego vamos a crear una nueva instancia de S3 y el cual espera que le pasemos las llaves de acceso que tenemos en nuestras variables de entorno:

var s3 = new aws.S3({

accessKeyId: config.aws.accessKey,

secretAccessKey: config.aws.secretAccessKey

})

Por ultimo este espera que creemos el storage donde estará todo nuestro código de configuración del multerS3 que importamos hace rato:

var storage = multerS3({

s3: s3,

bucket: 'nombre del bucket que creamos',

acl: 'public-read',

metadata: function(req, file, cb){

cb(null, {fileName: file.fieldname})

},

key: function(req, file, cb){

cb(null, +Date.now() + '.' + ext(file.originalname))

}

})

*• La primera clave es la instancia de S3 que creamos.  
 • bucket es el nombre del bucket que creamos en la pagina de AWS  
 • ACL: este nos determina el tipo que serán nuestros archivos que subamos, pueden ser privados o públicos, esto significa que los archivos que definamos como públicos-read cualquiera podrá verlos pero no podrá modificarlos o sobrescribirlos.  
 • metadata: este nos define el nombre del archivo, le pasamos el cb y en clave el nombre del archivo. (Access Control List).  
 • key: este nos define la fecha en el que se subirá este archivo. Lo mismo que el anterior, le pasamos el cb y este manipula   
 el archivo que le llega por parámetro.*

Luego hacemos la instancia de upload con nuestro storage:

var upload = multerS3({ storage: storage }).single('picture');

Y por último esta será nuestra ruta POST:

app.post('/api/pictures', function (req, res) {

upload(req, res, function (err) {

if (err){

return res.send(500, "Error uploading file");

}else {

res.send('File uploaded');

}

})

})

Ahora podemos corer nuestro Proyecto, pero primero vamos a aprender a crear variables de entorno ya que estamos usando estas variables en el proyecto:

Vamos a la página de AWS y en la parte superior izq. Seleccionamos Security Credentials y luego entramos a Access Keys donde podemos generar nuestra Key de acceso. Este cuando la genera nos entrega el AccessKey y el SecretAccessKey, es importante saber que esta SecretAccessKey solo la podremos ver en ese momento ya que no podemos volverla a ver para esto seria bueno guardarla, en cambio la AccessKey si la podemos consultar fácilmente.

Podemos crear para fines de prueba un archivo env.sh donde vamos a crear las variables de entorno. Este es un ShellScript lo podemos correr según nuestro Sistema operativo. Ya que usamos Windows vamos a hacer un pequeño tutorial para hac erlo en Windows:

•Click derecho MiPC  
 •Configuracion avanzada del sistema  
 •Variables de entorno  
 •En la sección variables del sistema nueva ingresar nombre y valor de cada una de las variables  
 •Aceptar  
 •Aceptar

**Probar que este correctamente ingresada** set NOMBRE\_VARIABLE

Ejemplo:

Imprime el valor que se le asigno a la variable  
 echo %AWS\_ACCESS\_KEY%  
 AWERGF45F857AWFFQFQ45

Si la variable no esta correctamente ingresada se imprimirá:  
 %AWS\_ACCESS\_KEY%

Por ultimo ya podremos correr nuestro comando del server y ver si funciona nuestro proyecto y si sube las imágenes a S3.

**Ahora vamos a crear el módulo de bases de datos que tenemos en nuestro esquema:**

Como agregar Standard (Linter) a nuestro código en JavaScript

Vamos a dejar a un lado nuestro server para trabajar en nuestro módulo de base de datos, que por cierto usara un linter.

Un linter es una herramienta que nos ayudan a seguir las buenas prácticas o guías de estilo de nuestro código fuente. En este caso, para JavaScript, usaremos uno llamado standard, lo instalamos por medio de NPM usando:

npm install standard --save-dev

Para consultar las reglas que propone el linter, lo puedes hacer desde:

<https://www.npmjs.com/package/standard>

Ahora vamos a crear una carpeta por fuera de platzigram para nuestro módulo de Bases de datos. Luego iniciamos un proyecto de npm con el comando de siempre:

npm init

Ahora que ya tenemos instalado standard, vamos a nuestro package.json y vamos a crear un script para correr nuestro linter cuanto vayamos a codear nuestro proyecto. Este es el script:

"scripts": {

"lint": "standard"

}

Ya luego podemos ejecutar nuestro script y este nos arrojara los errores que tengamos en el código.

Recordar: Siempre dejar una línea en blanco después de todo el código o si no nos arrojara error el linter.

Si queremos usar completamente las funcionalidades de EcmaScript 6 podemos instalar el transpilador de babel para poder usar todas estas nuevas funcionalidades del lenguaje en el 2015.

npm install --save-dev babel-eslint

En nuestro package.JSON creamos una clave “standard” y le pasamos lo siguiente:

"standard": {

"parser": "babel-eslint"

}

Testing en JavaScript usando AVA

Continuando con el módulo de base de datos, vamos a requerir de una librería de testing para asegurar que cada parte de nuestra aplicación este funcione correctamente.

Usaremos AVA, ya que la comunidad y soporte a la misma ha venido a crecer de manera interesante.

Para AVA necesitamos:

• Declararlo en nuestro archivo package.json  
 • Un archivo de test-js dentro de una carpeta llamada test  
 • Inclusive, podemos hacer test anónimos con async/awake, trabajar con promesas y hacer un test sin importar en que co diciones del código se presente.

Vamos a usar el **test driven development** que consiste en desarrollar primero las pruebas unitarias para el código y luego refactorizamos con el código que hacemos. Para ayudar esta tarea vamos a usar un **Test Runner,** que nos permite ejecutar los test que nosotros definamos. Para esto usamos AVA.

AVA es concurrente, quiere decir que usa todo los core de nuestro PC y distribuye los test en distintos procesos lo cual hace que nuestros test se ejecuten rápidamente.

Con este instalamos AVA:

npm install ava --save-dev

Luego vamos a nuestro package.JSON y vamos a crear nuestro script para correr lint y nuestros test:

"test": "npm run lint && ava"

Recordar: AVA siempre buscara por defecto los test en la carpeta “test”.

Este es el código del archivo utils-test.js:

'use strict'

const test = require('ava')

test('this should pass', t => {

t.pass()

})

test('this should fail', t => {

t.fail()

})

test('it should support async/awit', async t => {

let p = Promise.resolve(42)

let secret = await p

t.is(secret, 42)

})

•Requerimos AVA  
• test() necesita por parámetro un nombre, luego un callback que se ejecutara con los resultados del test.  
•Una cosa interesante de AVA es que puede correr test asincrónicos, soporta async/await o promesas.

Prueba unitaria para validar un array

Los test nos ayudan a crear esos casos de uso que esperamos tengan nuestra aplicación. En este ejemplo podemos predecir y comparar un arreglo con “hashtags” que tendrá una imagen de nuestra aplicación.

La aserción que usamos para verificar cada elemento de nuestro arreglo seria t.deepEqual.

Creamos un archivo utils en otro directorio, que contiene efectivamente esos valores de prueba que vamos a comparar y que nos permite usar este módulo en más partes de la aplicación.

Primero vamos a crear nuestros casos de prueba según los escenarios que nos pueda devolver la imagen, por ejemplo, que nos devuelva una imagen hashtags escritos de diversas formas e.g #Platzi #AwEsOmE ##yes #100 #Hia, lo que queremos es que este nos devuelva estos hashtags sin él # y en minúsculas. Vamos a ver el código de los diversos casos de prueba:

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

test('extracting hashtags from text', t => {

let tags = utils.extractTags('a #picture with tags #AwEsOmE #Platzi #AVA and #100 ##yes')

t.deepEqual(tags, [

'picture',

'awesome',

'platzi',

'ava',

'100',

'yes'

])

tags = utils.extractTags('a picture with no tags')

t.deepEqual(tags, [])

tags = utils.extractTags()

t.deepEqual(tags, [])

tags = utils.extractTags(null)

t.deepEqual(tags, [])

})

En realidad es un caso no mas pero con el método .deepEqual vamos a comparar lo que recibe el caso y entrara a evaluar si es igual a lo que plantea esa comparación. Siguiente a esto vamos a definir nuestro modulo que nos va a devolver los resultados que esperan los test para pasar:

•Creamos una carpeta en la raíz llamada lib y luego creamos nuestro modulo para exportar.

'use strict'

// Devolvemos un objeto con la función que nos retorna el array con los hashtags

// que tenga nuestra foto

const utils = {

extractTags

}

function extractTags (text) {

return []

}

module.exports = utils

Uso de expresiones regulares para nuestras pruebas unitarias

Las expresiones regulares nos ayudan bastante para coincidir o comparar los valores de cualquier valor de entrada.

Regexr.com, es una herramienta que nos facilita saber que reglas necesitamos para nuestra expresión regular. Esta regla, puede ser configurada dentro de nuestro .match()

Igualmente, normalizamos el flujo de datos recordando el método toLowerCase().

**Expresiones regulares:**

Las expresiones regulares son patrones utilizados para encontrar una determinada combinación de caracteres dentro de una cadena de texto. En JavaScript, las expresiones regulares también son objetos. Estos patrones son utilizados a través de los métodos exec y test de RegExp, así como los métodos match, replace, search y split de String. En este capítulo se describe el uso y funcionamiento de las expresiones regulares en JavaScript.

Consultar la documentación de estas expresiones:

<https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_Expressions>

Ahora vamos a construir nuestro método que nos traiga limpito nuestros #hashtags, lo que queremos es hacer que nos devuelva los hashtags sin el carácter # y todos en lowercase. Asi construimos nuestros métodos.

'use strict'

// Devolvemmos un objeto con la funcion que nos retorna el array con los hashtags

// que tenga nuestra foto

const utils = {

extractTags

}

function extractTags (text) {

if (text == null) return []

let matches = text.match(/#(\w+)/g)

if (matches === null) return []

matches = matches.map(normalize)

return matches

}

function normalize (text) {

text = text.toLowerCase()

text = text.replace(/#/g, '')

return text

}

module.exports = utils

• match y replace son métodos de los strings, estos nos reciben por parámetro una expresión regular, match buscara todas las coincidencias en el texto y las devolverá en un arreglo, replace también buscara las coincidencias con la expresión regular que le pasemos pero este remplazara esa coincidencia por e valor que nosotros le definamos como segundo parámetro, replace también nos retorna el array con los remplazos.

• Comparamos en el primer if con == y no con === ya que con == comparamos si es null y también comparara si este viene como indefinido, entonces seria como una comparación doble en una sola línea.

Introduccion a RethinkDB

RethinkDB es una base de datos pensada para funcionalidades realtime. Por tener bases de JavaScript en su core, la hace bastante compatible con proyectos bajo ese lenguaje.

Para instalar RethinkDB, basta con descargar el instalador para tu plataforma en:

<https://www.rethinkdb.com/docs/install/>

Una vez instalado RethinkDB, podemos iniciarlo con el comando:

rethinkdb (mac / linux)

C:\Users\TuUsurio\RethinkDB\>rethinkdb.exe

( Windows)

Podemos consultar la documentación en:

<https://www.rethinkdb.com/docs/guide/javascript/>

Una de las mayores ventajas que tiene RethinkDB es su administrador, donde puedes encontrar todas tus bases de datos, obtener estadísticas de las mismas, y hacer consultas usando su sintaxis ReQL.

La ventaja de la sintaxis de ReQL es que la sintaxis es muy similar a JavaScript, y los resultados de todas las consulta que se ejecuten desde el administrador se implementan igual desde node.js

RethinkDB usa la sintaxis ReQL que nos entrega los resultados en formato JSON y además nos permite encadenar los Querys asi como si fuera JS, podemos traer una tabla y luego de esas tablas filtrar un usuario y asi podríamos seguir encadenando las consultas sin nungun problema.

RethinkDB corre por defecto en el puerto 8080 ahí podemos ver su interfaz y podemos hacer querys desde su consola del navegador.

Accediendo a nuestra base de datos RethinkDB desde Node.js

Acceder a nuestra base de datos es muy fácil gracias a la API de RethinkDB, Lo primeroque necesitamos estructurar son las opciones de host, port y nombre.

Despues, la conexión puedes hacerla con callback o con promesas, esta segunda opción es la que recomendamos.

Para mejorar el flujo-control de nuestra aplicación, usaremos co, mejorando la sintaxis y teniendo en cuenta yield para pausar la ejecucion hasta que se resuelva la promesa.

Sobreescribimos la clase Promise que tiene por defecto ECMAScript por una librería llamada bluebird, permitiendo conectarnos como callback o como promesa (Interfaz Hibrida) a nuestra clase.

Vamos a crear nuestra clase de coneccion, esta mierda no la entendí ni un culo. **Repasar los putos generadores y las promesas y las funciones asincrónicas.**

'use strict'

const r = require('rethinkdb')

const co = require('co')

const Promise = require('bluebird')

const defaults = {

  host: 'localhost',

  port: 28015,

  db: 'platzigram'

}

class Db{

  constructor (options) {

    options = options || {}

    this.host = options.host || defaults.host

    this.port = options.port || defaults.port

    this.db = options.db || defaults.db

  }

  connect (callback) {

    this.connection = r.connect({

      host: this.host,

      port: this.port

    })

    let db = this.db

    let connection = this.connection

    let setup = co.wrap(function\* (){

      let conn = yield connection

      let dbList = yield r.dbList().run(conn)

      if (dbList.indexOf(db) === -1) {

        yield r.dbCreate(db).run(conn)

      }

      let dbTables = yield r.db(db).tableList().run(conn)

      if (dbTables.indexOf('images') === -1){

        yield r.db(db).tableCreate('images').run(conn)

      }

      if (dbTables.indexOf('users') === -1){

        yield r.db(db).tableCreate('users').run(conn)

      }

      return conn

    })

    return Promise.resolve(setup()).asCallback(callback)

  }

}

module.exports = Db

Para entender esta mierda, buscar los generadores, promesas y demás mierdas asincrónicas.

Las **Promesas** nos retornan una promesa resuelta, este recibe por parametro dos cosas, resolve y reject, usamos resolve cuando el resultado de la promesa es exitoso, el caso contrario seria para reject. Podemos manejar estos resultado con los metodos .then() y .catch() que ambos reciben una funcion por parametro con el parametro de resultado de la promesa. Este seria un ejemplo de una promesa.

var p = new Promise((resolve, reject) =>{

  setTimeout(function() {

    resolve('Me resolvi asincronamente')

  }, 2000);

})

p.then((resultPromise) =>{

  console.log(`${resultPromise} y me agregaron despues texto.`)

})

.catch((erro) =>{

  console.log(err)

})

Las funciones generadoras nos permiten generar resultado o codigo cuando nosotros queramos, estas funciones generadoras son instancias de los iteradores en las cuales vamos a poder hacer uso del metodo next() para iterar nuestra funcion. Podemos hacer uso de la palabra yield para detener nuestro codigo en un punto especifico y retornar valores en esa linea en especifico. Vamos a ver un ejemplo de estas funciones:

function\* gen(a, b) {

  yield a + b;

  yield a - b;

  yield a \* b;

  yield a / b;

}

var generador = gen(3, 2)

generador.next();

En este caso podemos iterar 4 veces nuestra funcion ya que tenemos cuatro **yield** en nuestra funcion, cada next() que hagamos nos va a retornar el valor de el primer **yield** que encuentre, luego volvemos a llamar a next() y retorna el valor del siguiente **yield** y asi hasta no tener mas **yield**. Cuanto hacemos uso de **yield** y no tenemos mas que iterar, este nos retorna otro resultado con la propiedad **value** en undefined y **done** en true. Mientras haya yield por iterar este nos devuelve **value** con el valor a retornar y **done** en false ya que no se ha finalizado el generador.

Luego aparece una forma de hacer generadores de forma mas sencilla, esta es el **async/await**, que nos permite definir una funcion como asincrona y hacer uso de la palabra **await** para detener la ejecucion de la funcion y esperar a que se resuelva la promesa anterior. Ahora veremos un ejemplo de la implementacion de **async/await** que espera hasta que se resuelva la promesa de la funcion anterior y ahí si sigue con el flujo normal de su codigo.

function promiseReturn (texto){

  return new Promise((resolve, reject) => {

    setTimeout(function() {

      resolve(`Esto es lo que quiero que imprima: ${texto}`)

    }, 3500);

  })

}

async function myFunctionAsync(){

  let data = await promiseReturn('Esta es mi promesa para Async/await prros')

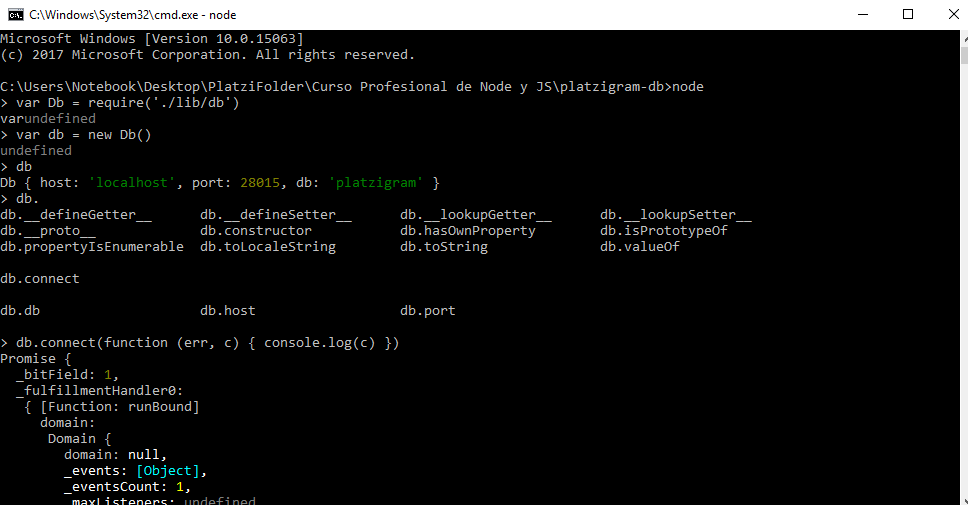
  console.log(data)

  let suma = 3 + 3;

  console.log(suma)

}

Ahora continuando con el flujo de esta clase, ahora vamos a realizar los siguientes comandos para ejecutar nuestra clase que implementamos en node para crear la coneccion a la base de datos:



• Podemos en lugar de pasarle un **callback** una **promise** y este nos ejecutara la coneccion de igual manera ya que estamos manejando la coneccion de manera hibrida con bluebird y con la implementacion de rethinkDB.

Ahora ya entiendo todo lo de funciones generadoras y async/await y promesas, ESTOY MAS QUE FELIZ!

Creacion de test para RethinkDB y almacenamiento de ruta de imágenes

Vamos a crear un test que soporte tanto la conexión a base de datos, como el almacenamiento de una imagen. Para ello, necesito

agregar una librería de creación de IDs que dará nombres únicos e identificados a nuestras bases de datos de pruebas que instaciamos.

Por cierto, este test será asíncrono, tendrá una nueva aserción llamad t.is, para asegurarnos que contendrá los métodos que esperamos y un mensaje de feedback que nos diga que la base de datos se conecta/desconecta correctamente.

Vamos a usar una librería que nos permitira generar UUID en base62 aleatorios, estos los utilizamos para asignarselos a las base de datos de prueba que vamos a crear en los test. Esto para que nunca hagamos las pruebas sobre nuestra base de datos de produccion.

npm install uuid-base62 --save

Luego requerimos el modulo que acabamos de instalar y lo podemos usar. Si tenemos problemas con lo que son las UUID estas son **Indentificadores unicos universales**, si queremos saber mas sobre la definicion de estos podemos revisar aca la informacion:

<https://es.wikipedia.org/wiki/Identificador_%C3%BAnico_universal>

Por otro lado si queremos saber mas sobre el modulo de npm aquí esta la documentacion:

<https://www.npmjs.com/package/uuid-base62>

**Recordar:** revisar la carpeta de favoritos que esta en google chrome, en esta se encuentran paginas con informacion util sobre el curso.

Ahora vamos a crear nuestro modulo de prueba para nuestra base de datos, el codigo quedo asi:

'use strict'

const test = require('ava')

const uuid = require('uuid-base62')

const Db = require('../')

const r = require('rethinkdb')

const dbName = `platzigram\_${uuid.v4()}`

const db = new Db({ db: dbName })

test.before('setup databese', async t => {

await db.connect()

t.true(db.connected, 'should be connected')

})

test.after('disconnect database', async t => {

await db.disconnect()

t.false(db.connected, 'should be disconnected')

})

test.after.always('cleanup database', async t => {

let conn = await r.connect({})

await r.dbDrop(dbName).run(conn)

})

test('save image', async t => {

t.is(typeof db.saveImage, 'function', 'saveImage is function')

let image = {

url: `https://platzigram.test/${uuid.v4()}.jpg`,

likes: 0,

liked: false,

user\_id: uuid.uuid()

}

let created = await db.saveImage(image)

t.is(created.url, image.url)

t.is(created.likes, image.likes)

t.is(created.liked, image.liked)

t.is(created.user\_id, image.user\_id)

t.is(typeof created.id, 'string')

t.truthy(created.createdAt)

})

1. Requerimos nuestra clase de esta forma ya que creamos un archivo **index.js** que es el punto de entrada de nuestro proyecto, tal cual esta definido en nuestro **package.json**. El contenido de este archivo es solo una exportacion de nuestra clase, por eso podemos hacer el require de esta forma. Este es el codigo del archivo:

module.exports = require('./lib/db')

1. Luego creamos la instancia de esta clase con el nombre y el UUID aleatorio.
2. AVA nos permite usar “hooks” o funciones que se ejecutaran antes o despues de los test. Creamos dos, uno para antes el cual nos conectara a la base de datos y el otro para despues que nos desconecte de la base de datos.
3. Luego vamos a crear estos metodos de connect y disconnect en nuestra clase. Este seria el codigo que agregariamos:

this.connected = true //Este va dentro de el metodo de setup

disconnect (callback) {

if (!this.connected) {

return Promise.reject(new Error('not connected')).asCallback(callback)

}

this.connected = false

return Promise.resolve(this.connection)

.then((conn) => conn.close())

}

1. Siguiendo con los “hooks” podemos crear uno que se ejecute siempre despues de cualquier test sin importar que los test fallen este siempre se ejecutara. En este caso lo usamos para borrar la base de datos que creamos como ejemplo para nuestro test.
2. Ahora vamos a crear un test para pobrar nuestras imágenes. Vamos a crear un objeto con lo que esperamos que compare y con lo que nos devolvera nuestra clase que se conectara a nuestra base de datos.

Ahora vamos a ver como implementar el metodo para almacenar una imagen en la base de datos.

Almacenamiento de rutas de imágenes en RethinkDB

Primero una acotación, no vamos a guardar directamente nuestras imágenes en RethinkDB, sería algo contraproducente almacenar un binario (imagen) debió a su capacidad realtime.

Lo que si vamos a almacenar, es precisamente la URL de donde estará almacenada esa imagen, que de hecho, será por medio de Amazon S3, aquí nos enfocamos en entender cómo será esa inserción a nuestra base de datos con node, mediante los métodos .table() y .insert() respectivamente.

Este es el codigo que vamos a crear dentro de nuestra clase para poder hacer el insert en nuestra base de datos con la url donde estara alojada nuestra foto:

saveImage (image, callback) {

if (!this.connected) {

return Promise.reject(new Error('not connected')).asCallback(callback)

}

let connection = this.connection

let db = this.db

let task = co.wrap(function \* () {

let conn = yield connection

image.createdAt = new Date()

let result = yield r.db(db).table('images').insert(image).run(conn)

if (result.errors > 0) {

return Promise.reject(new Error(result.first\_error))

}

image.id = result.generated\_keys[0]

return Promise.resolve(image)

})

return Promise.resolve(task()).asCallback(callback)

}

Creamos una corutina y las resolvemos con las promesas.

Ahora veamos como hacer la inserccion de los likes en nuestra foto.

Completando el test para nuestro objeto imagen

Estamos haciendo testing de cada funcionalidad de nuestra aplicación, en el capítulo anterior, vimos como nuestro modelo de una imagen contiene una URL, una descripción, “Hastags” o categorías, número de likes y el ID del usuario. Vamos a hacer una aserción en nuestro test para cada uno de esos campos de nuestro modelo.

Ahora vamos a crear nuestro codigo para esto:

test('save image', async t => {

t.is(typeof db.saveImage, 'function', 'saveImage is function')

let image = {

description: 'an #awesome picture with #tags #platzi',

url: `https://platzigram.test/${uuid.v4()}.jpg`,

likes: 0,

liked: false,

user\_id: uuid.uuid()

}

let created = await db.saveImage(image)

t.is(created.description, image.description)

t.is(created.url, image.url)

t.is(created.likes, image.likes)

t.is(created.liked, image.liked)

t.deepEqual(created.tags, ['awesome', 'tags', 'platzi'])

t.is(created.user\_id, image.user\_id)

t.is(typeof created.id, 'string')

t.is(created.public\_id, uuid.encode(created.id))

t.truthy(created.createdAt)

})

Aca agregamos unos campos para poner los tags y la descripcion, luego vamos a modificar nuestra clase Db para hacer la inserccion de nuestra imagen y tambien acortar el public\_id de nuestra imagen:

saveImage (image, callback) {

if (!this.connected) {

return Promise.reject(new Error('not connected')).asCallback(callback)

}

let connection = this.connection

let db = this.db

let task = co.wrap(function \* () {

let conn = yield connection

image.createdAt = new Date()

image.tags = utils.extractTags(image.description)

let result = yield r.db(db).table('images').insert(image).run(conn)

if (result.errors > 0) {

return Promise.reject(new Error(result.first\_error))

}

image.id = result.generated\_keys[0]

yield r.db(db).table('images').get(image.id).update({

public\_id: uuid.encode(image.id)

}).run(conn)

let created = yield r.db(db).table('images').get(image.id).run(conn)

return Promise.resolve(created)

})

return Promise.resolve(task()).asCallback(callback)

}

El codigo se entiende facilmente asi que no hare la explicacion de este ya que son cosas vistas, pero basicamente se agrego no mas el acortamiento del id.

Creacion de un fixture para test de peticion de imágenes

Un fixture, en el testing, es una práctica que nos ayuda a tener datos muy cercanos a los esperados y darnos todo el contexto necesario para el test, aislando totalmente la dependencia de datos reales de nuestra aplicación.

También vamos a agregar otro método encargado de guardar si una imagen fue marcada como favorita o no. Notaremos que a nivel de estructura es muy parecido a lo que hicimos al guardar una URL de almacenamiento. Repasamos el concepto de corutina, promesa y testing con AVA

Vamos a crear un fixture en la carpeta test, ahí guardaremos los datos que nos serviraran para nuestros test. Asi queria el codigo de este fixture:

'use strict'

const uuid = require('uuid-base62')

const fixtures = {

getImage () {

return {

description: 'an #awesome picture with #tags #platzi',

url: `https://platzigram.test/${uuid.v4()}.jpg`,

likes: 0,

liked: false,

user\_id: uuid.uuid()

}

}

}

module.exports = fixtures

Ahora vamos a implementar nuestro test en el archivo de test, asi quedaria nuestro codigo:

test('like image', async t => {

t.is(typeof db.likeImage, 'function', 'likeImage is a function')

let image = fixtures.getImage()

let created = await db.saveImage(image)

let result = await db.likeImage(created.public\_id)

t.true(result.liked)

t.is(result.likes, image.likes + 1)

})

Ahora vamos a crear nuestro metodo de likeImage() en nuestra clase de base de datos, el codigo es similar al metodo de saveImage() solo que cambia el contenido de nuestra corutina. Asi quedaria nuestro codigo:

likeImage (id, callback) {

if (!this.connected) {

return Promise.reject(new Error('not connected')).asCallback(callback)

}

let connection = this.connection

let db = this.db

let imageId = uuid.decode(id)

let task = co.wrap(function \* () {

let conn = yield connection

let image = yield r.db(db).table('images').get(imageId).run(conn)

yield r.db(db).table('images').get(imageId).update({

liked: true,

likes: image.likes + 1

}).run(conn)

let created = yield r.db(db).table('images').get(imageId).run(conn)

return Promise.resolve(created)

})

return Promise.resolve(task()).asCallback(callback)

}

Traemos el id que recibimos por parametro que es el **public\_id** que esta en base62, pero nosotros queremos el id privado de la imagen asi que lo que hacemos es decodificar este id que esta codificado y nos entrega el codigo que aparece registrado en la base de datos.

Luego en la corutina obtenemos la imagen como esta y luego hacemos un update a esta imagen sumandole la cantidad de likes que tenia antes y seteando el atributo **liked** en true. Luego obtenemos de nuevo la imagen pero ya actualizada y la retornamos para que haga el test con AVA.

Peticion de la imagen desde base de datos

Hemos agregado métodos para agregar elementos a nuestra base de datos, agregamos uno adicional que será la petición de una imagen, por su ID, dentro de la misma. Repetimos los pasos anteriores de corutina, promesa y testing con AVA

Ahora veamos como seria la implementacion de nuestro test en el archivo de test que hemos venido escribiendo:

test('get image', async t => {

t.is(typeof db.getImage, 'function', 'getImage is a function')

let image = fixtures.getImage()

let created = await db.saveImage(image)

let result = await db.getImage(created.public\_id)

t.deepEqual(created, result)

})

Realizamos un paso casi similar al test anterior. Solo que comparamos la imagen que creamos con el resultado de la busqueda por el id de esta misma.

Ahora que ya tenemos nuestro test creado vamos a implementar nuestro metodo de getImage() en nuestra clase Db:

getImage (id, callback) {

if (!this.connected) {

return Promise.reject(new Error('not connected')).asCallback(callback)

}

let connection = this.connection

let db = this.db

let imageId = uuid.decode(id)

let task = co.wrap(function \* () {

let conn = yield connection

let image = yield r.db(db).table('images').get(imageId).run(conn)

return Promise.resolve(image)

})

return Promise.resolve(task()).asCallback(callback)

}

Entonces es similar que la implementacion del metodo de likeImage() pero aquí solo obtenemos la imagen con el **id** que no llega por parametro, decodificamos este **id** primer y luego retornamos la imagen.