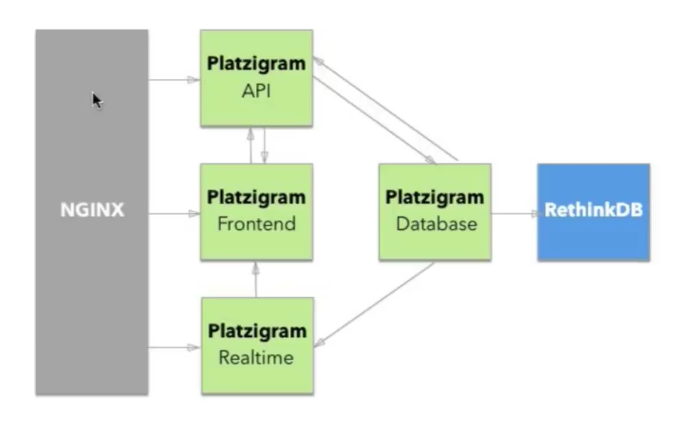
**CURSO PROFESIONAL DE NODE.js y JS**

**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.

Este es el esquema de nuestra aplicación:



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')

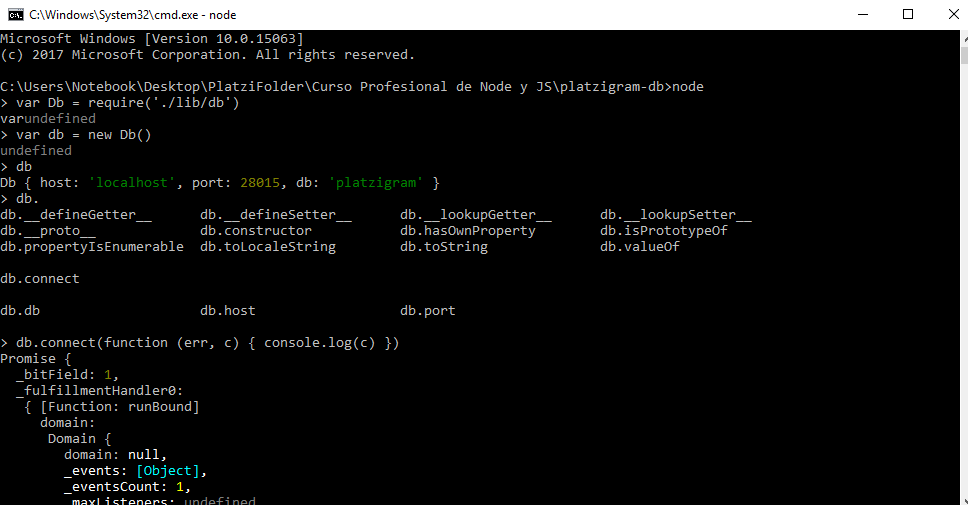
  console.log(data)

  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.

Consulta de todas las imágenes de nuestra base de datos

Nuestro caso ideal es consultar “n” imágenes en nuestra base de datos, para ello, vamos a construir nuestro test pensando en una lista de imágenes y en nuestro backend, crearemos ese mismo método de consulta.

Para lograr esto, implementamos un arreglo de promesas, donde fácilmente, las invocamos por medio de un map.

Del lado de RethinkDB, el nuevo query que utilizamos es .orderBy, eligiendo como parámetro el campo createdAt. ( Listar del elemento más reciente, al menos reciente, según cuando fue creado)

Primero como un issue que nos salio durante la clase fue que cada vez que ibamos a correr un test siempre se ejecutaba antes de todos los test un **hook**, para crear la conexión, el problema era que cuando se corrian los test todos los test compartian el mismo contexto de la creacion de la base de datos, entonces nunca limpiabamos nuestras variables en la base de datos y se acumulaban en los siguientes test y no nos pasaban nuestros test. Lo que hicimos fue crear unos **hooks** que se ejecutaran antes y despues de cada uno de los test y no para todos los test. Este codigo quedo asi:

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

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

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

await db.connect()

t.context.db = db

t.context.dbName = dbName

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

})

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

let db = t.context.db

let dbName = t.context.dbName

await db.disconnect()

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

let conn = await r.connect({})

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

})

• Compartimos en el contexto de las funciones de los test la db y el dbName que se crean especificamente para cada uno de los test y asi mantenemos nuestros test aislados y no tendremos problemas de que acumulemos valores de otros tests

Luego lo que hacemos en los test individuales es guardar el contexto que estamos compartiendo para cada uno de los test independientes. Asi quedaria el ejemplo del test para listar todas las imágenes que tengamos en la base de datos:

test('list all images', async t => {

let db = t.context.db

let images = fixtures.getImages(3)

let saveImages = images.map(img => db.saveImage(img))

let created = await Promise.all(saveImages)

let result = await db.getImages()

t.is(created.length, result.length)

})

Como el metodo **saveImages()** de nuestra clase **Db** nos retorna una promesa cada vez que llamamo ese metodo y como para este test vamos a recibir muchas promesas, vamos a resolverlas todas gracias al metodo **all()** de las promesas que nos entrega JS y luego si obtenemos estas y las comparamos.

Ahora veamos como seria el codigo de nuestro metodo **getImages()** de nuestra clase de **Db**:

getImages (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

let images = yield r.db(db).table('images').orderBy({

index: r.desc('createdAt')

}).run(conn)

let result = yield images.toArray()

return Promise.resolve(result)

})

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

}

• Convertimos el resultado de la consulta en un Array, ya que la base de datos nos entrega un puntero, el puntero es parecido a un iterador, a este puntero podemos aplicarle el metodo **next()** para que nos entregue el siguiente valor y asi. Pero como nosotros queremos hacer uso luego del metodo **map()** de los array y objetos los convertimos a un array.

• Tambien usamos el metodo de RethinkDB orderBy para que nos traiga a los elementos que tenga el indice **createdAt** y que no lo organice de forma descendente. Para crear el indice en la tabla vamos a crear el siguiente codigo en nuestra clase de Db cuando hacemos el set de nuestra Db de testeo. Este es el codigo: //Este va dentro de la corutina de **setup**

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

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

yield r.db(db).table('images').indexCreate('createdAt').run(conn)

}

Buenas prácticas de creación de usuarios en Backend (Encriptación con Hash)

La seguridad de los datos de nuestros usuarios debe ser prioridad en cualquier desarrollo que tengamos, existen muchas técnicas para encriptarlos en nuestra base de datos, sin embargo, para este ejemplo, usamos una técnica llamada Hash justo al algoritmo “sha256”.

Incluimos un fixture que agregara todo el contexto necesario a nuestro test, y también, nos aseguramos de implementar nuestro método en nuestros backend.

Para implementar este algoritmo **node.js** incluye ya un modulo de encriptacion con estos algoritmos llamado crypto, lo unico que hacemos es requerir este modulo y empezar a usarlo:

const crypto = require('crypto')

Luego procedemos a crear nuestra funcion que va a crear nuestra contraseña encriptada:

function encrypt (password) {

let shasum = crypto.createHash('sha256')

shasum.update(password)

return shasum.digest('hex')

}

Esta seria la estructura basica del metodo de crypto que nos provee **node.js**. Debemos mirar la documentacion que nos provee **node.js**: <https://nodejs.org/api/crypto.html>

Luego vamos a crear nuestros test para grabar nuestro usuario con las contraseñas encriptadas por que no es buena practica guardar la contraseña en texto plano de nuestros usuario en la base de datos. En caso de que nuestra base de datos se vea comprometida vamos a tener la informacion encriptada y no hay datos que se puedan comprometer.

test('encrypt password', t => {

let password = 'foo123'

let encrypted = '02b353bf5358995bc7d193ed1ce9c2eaec2b694b21d2f96232c9d6a0832121d1'

let result = utils.encrypt(password)

t.is(result, encrypted)

})

Luego vamos a crear el test para nuestra base de datos cuando hagamos el save de nuestro usuario. Este es el codigo:

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

let db = t.context.db

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

let user = fixtures.getUser()

let plainPassword = user.password

let created = await db.saveUser(user)

t.is(user.username, created.username)

t.is(user.email, created.email)

t.is(user.name, created.name)

t.is(util.encrypt(plainPassword), created.password)

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

t.truthy(created.createdAt)

})

Y creamos nuestro metodo en la clase de **Db**, este seria el codigo:

saveUser (user, 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

user.password = utils.encrypt(user.password)

user.createdAt = new Date()

let result = yield r.db(db).table('users').insert(user).run

if (result.errors > 0) {

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

}

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

let created = yield r.db(db).table('users').get(user.id).run

return Promise.resolve(created)

})

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

}

Peticion de un usuario desde base de datos.

Ahora, haremos nuestra petición de un usuario, a la base de datos por su ID. Repetimos los pasos anteriores de corutina, promesa y testing con AVA.

Del lado de RethinkDB, el nuevo query que utilizamos es .getAll( ), eligiendo como parámetro el nombre de usuario.

Vamos a crear nuestro test para obtener nuestro usuario por su username.

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

let db = t.context.db

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

let user = fixtures.getUser()

let created = await db.saveUser(user)

let result = await db.getUser(user.username)

t.deepEqual(created, result)

})

Luego vamos a implementar nuestro metodo de **getUser()** que vamos a probar:

getUser (username, 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

yield r.db(db).table('users').indexWait().run(conn)

let users = yield r.db(db).table('users').getAll(username, {

index: 'username'

}).run(conn)

let result = yield users.next()

return Promise.resolve(result)

})

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

}

Vamos a crear un indice en nuestro en modulo de setup como hicimos para nuestras imágenes pero esta vez va a ser para buscar nuestros usuarios por una *secondary key*. Luego en la implementacion del metodo de **getUser()** vamos a agregar un **indexAwait()** para esperar a que se creen todos nuestros indices por la tabla y luego si empezamos a buscar por nuestro indice nuestro usuario.

Autenticacion de usuario mediante password

Este es nuestro primer método, dentro de nuestro módulo de [db.js](https://platzi.com/clases/node-2016/concepto/modulo-de-base-de-datos-con-rethinkdb/autenticacion-de-usuario-mediante-password/material/db.js) , que va a requerir 2 parametros: Usuario y password.

Para obtener el usuario, podemos hacer uso de la corutina de consulta de usuarios.

Para hacer la comparación del password, hacemos una comparación entre el texto ingresado y el valor encriptado guardado en la base de datos.

Vamos a crear nuestro test para la autenticacion de nuestros usuarios, este es el codigo:

test('authenticate user', async t => {

let db = t.context.db

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

let user = fixtures.getUser()

let plainPassword = user.password

await db.saveUser(user)

let success = await db.authenticate(user.username, plainPassword)

t.true(success)

let fail = await db.authenticate(user.username, 'foo')

t.false(fail)

})

Luego hacemos el mismo metodo para autenticar los usuarios que tenemos almacenados en nuestra base de datos:

authenticate (username, password, callback) {

if (!this.connected) {

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

}

let getUser = this.getUser.bind(this)

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

let user = yield getUser(username)

if (user.password === utils.encrypt(password)) {

return Promise.resolve(true)

}

return Promise.resolve(false)

})

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

}

Control de errores mediante excepciones (Try, Catch, Throws)

Es un buen momento para practicar lo aprendido mediante un refactor. La meta será, dirigir correctamente los errores de nuestros test, a algo más descriptivo mediante el uso de manejo de excepciones (Try, Catch, Throws ).

Aca solo manejamos los errores de nuestros test y en la implementacion de nuestros metodos en la clase de **Db**, si queremos ver como se manejaron ver la clase **Db** en el repositorio de github.

Consulta de imágenes en relacion a nombre de usuario

Queremos obtener sólo las imágenes creadas por cierto usuario. Para ello, primero hacemos la consultas de todas las imágenes disponibles y indexamos por el userID que esperamos. Por ser un arreglo, tendremos que considerar múltiples promesas.

Entonces, mediante rethinkDB, vamos a buscar todas las imágenes [.getAl](https://platzi.com/clases/node-2016/concepto/modulo-de-base-de-datos-con-rethinkdb/consulta-de-imagenes-en-relacion-a-nombre-de-usuar/material/.getAl) l() , usando como index el userID, y ordenando esos resultados de manera descendente.

**Hasta aquí nos funcionan los test, asi que de ahora en adelante voy a dejar el codigo de los tests tal cual como estan en la clase pero ya no los voy a correr en el proyecto.**

Ahora vamos a ver el codigo del test para listar las imágenes por id de usuario:

test('list images by user', async t => {

t.is(typeof db.getImagesByUser, 'function', 'getImagesByUse is a function')

let images = fixtures.getImages(10)

let userId = uuid.uuid()

let random = Math.round(Math.random() \* images.length)

let saveImage = []

for (let i = 0; i < images.length; i++) {

if (i < random) {

images[i].userId = userId

}

saveImage.push(db.saveImage(images[i]))

}

await Promise.all(saveImage)

let result = await db.getImagesByUser(userId)

t.is(result.length, random)

})

Resolvemos con un Promise.all ya que el arreglo de nuestras imágenes contienen el retorno de nuestro metodo de nuestra base de datos que nos devuelve las promesas pendientes.

Luego vamos a crear nuestro metodo de **getImagesByUser()** en nuestra clase de de base de datos **Db**:

getImagesByUser (userId, password, 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

yield r.db(db).table('images').indexWait().run(conn)

let images = yield r.db(db).table('images').getAll(userId, {

index: 'userId'

}).orderBy(r.desc('createdAt')).run(conn)

let result = yield images.toArray()

return Promise.resolve(result)

})

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

}

Creamos la consulta para nuestra funcionalidad y ademas debemos crear un indice nuevo en nuestro apartado de **setup()** para encontrar imágenes por el ID del usuario. Luego en la consulta con el metodo **getAll()** de RethinkDB le decimos que busque las imágenes por ese ID que es un indice en nuestra base de datos.

Aquí esta el codigo que se agrego en el apartado de **setup()** para crear el nuevo indice en la tabla images:

yield r.db(db).table('images').indexCreate('userId', { multi: true }).run(conn)

Aqui en **indexCreate()** le pasamos adicionalmete el atributo **multi: true** para que RethinkDB nos acepte este indice en multiples imágenes y para cuando traigamos nuestras imágenes nos traiga todas las imágenes que se relacionen con ese ID.

Consulta de imágenes en relacion a tag

Queremos obtener sólo las imágenes de acuerdo a un tag. Para ello, primero hacemos la consultas de todas las imágenes disponibles y filtramos por el tag que esperamos. Por ser un arreglo, tendremos que considerar múltiples promesas.

Entonces, mediante rethinkDB, vamos a buscar todas las imágenes [.getAl](https://platzi.com/clases/node-2016/concepto/modulo-de-base-de-datos-con-rethinkdb/consulta-de-imagenes-en-relacion-a-tag/material/.getAl) l() , usando como filtro el tag, y ordenando esos resultados de manera descendente. Ese filtro, puede ser referenciado, como un arrow function.

Aquí vamos a realizar igualmente como para los casos anteriorires, nuestros test y luego la implementacion de nuestro metodo en la clase de base de datos **Db**. Este es el codigo para el test:

test('list images by tag', async t => {

let db = t.context.db

t.is(typeof db.getImagesByTag, 'function', 'get ImagesByTag is a function')

let images = fixtures.getImages(10)

let tag = '#filterit'

let random = Math.round(Math.random() \* images.length)

let saveImages = []

for (let i = 0; i < images.length; i++) {

if (i < random) {

images[i].description = tag

}

saveImages.push(db.saveImage(images[i]))

}

await Promise.all(saveImages)

let result = await db.getImagesByTag(tag)

t.is(result.length, random)

})

Esta vez no pasamos un ID, pasamos un tag por el cual queremos filtrar nuestras imágenes.

Ahora este es el codigo de la implementacion en la base de datos:

getImagesByTag (tag, callback) {

if (!this.connected) {

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

}

let connection = this.connection

let db = this.db

tag = utils.normalize(tag)

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

let conn = yield connection

yield r.db(db).table('images').indexWait().run(conn)

let images = yield r.db(db).table('images').filter((img) => {

return img('tags').contains(tag)

}).orderBy(r.desc('createdAt')).run(conn)

let result = yield images.toArray()

return Promise.resolve(result)

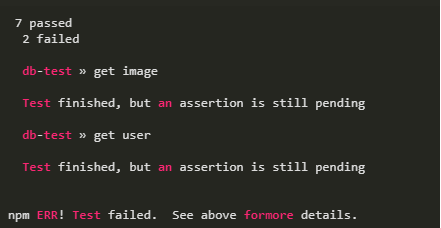
})

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

}

Aquí hicimos uso de una consulta de RethinkDB que nos permite **filtrar** las imágenes por un campo en especifico y que nos devuelvan las imágenes que **contengan** el **tag** por el que queremos filtrar. Lo demas es similar a lo que hemos venido trabajando.

Anteriormente estaba sufriendo de un error de que los test no pasaban y se quedeban “pensando” y nunca terminaban, este primer problema creo que fue por que tenia algo mal copiado, entonces remplace el codigo y luego ya terminaban los test. Pero luego de esto ya arrojaba que 2 test fallaban con el siguiente error:



Gracias a un aporte de un estudiante de la comunidad de platzi se resolvio, este fue su comentario y la solucion a esto:

“Me respondo gracias a la documentación de AVA en el apartado de las assertions: cuando se prueba una promesa se deba esperar a que la aserción se complete con la expresión await, la cual, hace que la ejecución de la función asíncrona haga una pausa, espere la resolución de la Promesa y reanude la ejecución de la función asíncrona cuando se resuelve el valor.”

Este es el codigo adjunto:

await t.**throws**(db.getUser('foo'), /not found/)

**await** t.throws(db.getImage('foo'), /not found/)```

**Recordatorio: Nunca dejar de mirar la documentacion de los proyectos. Cuando sea necesario echar siempre una miradita a estos ya que se encuentra casi siempre la solucion en estas.**

Preparando nuestro modulo para microservicios

Vamos a preparar nuestro módulo de base de datos para ser usado en cualquier lado de nuestra aplicación y esto será agregando un flag a nuestro constructor, de este módulo, aprendimos buenas prácticas, testing e implementación. Ya toda la persistencia de datos está lista.

Vamos hacer que nuestro setup de la clase de base de datos **Db** no configure la base de datos cada vez que se conecte, lo vamos a hacer solo para las pruebas pero cuando ya tenemos garantizado que la base de datos ya esta correctamente configurada no lo hacemos ya que este proceso es poco optimo. Entonces vamos a modificar nuestro codigo y el codigo de los test:

constructor (options) {

options = options || {}

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

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

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

this.setup = options.setup || false

}

Pasamos una opcion para hacer setup, y que por defecto tendra el valor de false.

if (!this.setup) {

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

}

Luego si esta en false el valor de setup simplemente retornamos la conexcion como callback ya resuelto.

Si lo anterior esta en true pues entraria a hacer todo el setup de la base de datos de nuevo.

Ahora vamos a ver el codigo que cambio en los test de nuestra base de datos:

test.beforeEach('setup database', async t => {

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

const db = new Db({ db: dbName, setup: true })

await db.connect()

t.context.db = db

t.context.dbName = dbName

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

})

Agregamos la opcion de setup en true para que configure una nueva base de datos para prueba. Y listo.

Terminamos nuestro modulo de base de datos, aprendimos a encriptar contraseñas, a crear ID unicos para usuarios e imágenes, a crear tests unitarios para cada funcionalidad y luego implementarlos, test asincronos, Querys con el lenguaje de Querys de RethinkDB, manejo de expresiones regulares y uno que otro metodo de los array y los Strings.

Ahora veamos que son los microservicios y como implementarlos.

Introduccion a microservicios

Recordando un poco la estructura de nuestra aplicación, hemos terminado con el primer bloque (Platzigram Database) y continuaremos con el API, pero: ¿Porqué está dividido en bloques?, porque estamos implementando algo llamado microservicios.

Dependiendo del tamaño de las aplicaciones, nosotros podemos separar las interacciones de nuestra solución en diversos módulos independientes entre sí, quiere decir, si una parte de tu aplicación falla, esto no va a tirar o romper todas las demás.

De manera inherente, aprendemos a crear aplicaciones escalables y modulares.

Una librería que nos ayudará como orquestador de nuestros microservicios se va a llama micro por zeit.

Creando el proyecto desde cero, vamos a incluir lo que anteriormente hemos usado: Un linter, test unitario (con AVA y test-listen) y lo nuevo, la base de cómo se instancia un microservicio.

Vamos a crear nuestro proyecto de nuevo en una nueva carpeta y la inicializamos con npm init. Luego vamos a instalar nuestras dependencias de desarrollo:

npm install --save-dev ava standard babel-eslint test-listen

Luego vamos a instalar las siguientes dependencias:

npm install request-promise micro uuid-base62 babel-runtime request --save

**Test-listen** nos permite ejecutar los test para los microservicios, asi como haciamos con los test en la base de datos, test-listen nos va a levantar el microservicio de prueba para hacer el test y luego lo desechara cuando se acabe el test. Este corre los test con ava y los une con nuestro microservicios con **micro**.

**Request-promise** nos permite realizar request HTTP con promesas, en lugar de ejecutarlas con callbacks normales, este se adapta bien a nosotros ya que nosotros siempre estamos devolviendo promesas.

**Micro** es el que nos permite crear microservicios usando **Async/Await** y de forma facil, en este vamos a realizar toda la creacion de nuestras rutas y endpoint de nuestra REST API.

Por ultimo usamos el **babel-runtime** simplemente ya que micro necesita de esta dependencia.

Luego vamos a definir un test de prueba:

'use strict'

import test from 'ava'

import micro, { send } from 'micro'

import uuid from 'uuid-base62'

import listen from 'test-listen'

import request from 'request-promise'

test('GET /:id', async t => {

let id = uuid.v4()

let srv = micro(async (req, res) => {

send(res, 200, { id })

})

let url = await listen(srv)

let body = await request({ uri: url, json: true })

t.deepEqual(body, { id })

})

test.todo('POST /')

test.todo('POST /:id/like')

• Micro recibe una funcion asincrona por parametro que es el request listener, luego hacemos uso de la funcion send de micro al cual le pasamos como argumento la respuesta, el status code y el contenido de la respuesta.

**Tip**: Si la clave y el valor tienen el mismo nombre podemos evitar la duplicidad al escribir estos y solo escribimos una vez el nombre como en el caso del contenido de nuestras respuesta en el send.

• Listen nos va a retornar una url con puerto del servidor que estamos corriendo en micro.

•Luego hacemos la peticion HTTP y la resolvemos tambien con **await**, le pasamos la url y le indicamos en que formato nos devuelva el resultado

Creacion de rutas para microservicios

Los microservicios no son tan complejos como un framework, por lo tanto, para la creación de rutas y middlewares necesitamos el apoyo de otras librerías.

**HttpHash,** nos ayuda a crear las rutas de nuestros microservicio, la cual, por el momento, son 2: Una para obtener una imagen y la otra para subir una imagen.

No olvidemos la implementación de Test unitarios gracias a AVA.

Micro por defecto espera que nosotros exportemos una funcion asincrona que es la que el escucha y luego empieza a servir y que recibe nuestros objetos de request y response. Entonces en esta funcion es donde deberiamos ejecutar la logica para cuando nos llega una peticion para la ruta A o una peticion para la ruta B.

export default async function main (req, res) {

}

Ahora vamos a instalar **HttpHash**:

npm install --save http-hash

Podemos ver la documentacion de esta librería para apoyarnos mas:

<https://github.com/Matt-Esch/http-hash>

Veamos como es la implementacion de http-hash, como definimos nuestras rutas y les pasamos un handler asincrono:

import HttpHash from 'http-hash'

import { send } from 'micro'

const hash = HttpHash()

hash.set('GET /:id', async function getPicture (req, res, params) {

send(res, 200, params)

})

export default async function main (req, res) {

let { method, url } = req

let match = hash.get(`${method.toUpperCase()} ${url}`)

if (match.handler) {

try {

await match.handler(req, res, match.params)

} catch (e) {

send(res, 500, { error: e.message })

}

} else {

send(res, 404, { error: 'route not found' })

}

}

• Definimos con set la ruta y le pasamos un handler asincrono, esto no lo podemos hacer con express.  
 • Luego en la funcion que vamos a exportar hacemos el matcheo de la ruta que nos ingresen, pero antes seteamos los los valores que nos llegan por el parametro request, que estos seran la url y el metodo que ejecutan el request.  
 • Luego verificamos si la ruta existe con una condicion y luego como manejamos todo en nuestra Base de datos con Promesas tenemos que resolver esto con **Async/Await** y la manera de capturar errores en esto es con el viejo **try/catch**

Ahora vamos a correr nuestros test pero primero tenemos que agregar unas funcionalidades de babel ya que ava no nos soporta exportaciones de modulos con el ES2015 entonces vamos a instalar las siguientes dependencias:

npm install --save babel-plugin-transform-async-to-generator babel-preset-es2015 babel-plugin-transform-runtime

Luego instalamos esta otra funcionalidad:

npm install --save-dev babel-register

Luego de esto modificamos nuestro **package.json** para decirle a ava y a babel que use estas funcionalidades que acabamos de instalar:

"babel": {

"presets": ["es2015"],

"plugins": [

"transform-runtime",

"transform-async-to-generator"

]

},

"ava": {

"require": [

"babel-register"

]

},

Por ultimo modificamos nuestro test ya que lo teniamos corriendo el servidor directamente en el test, este es el codigo del test:

test('GET /:id', async t => {

let id = uuid.v4()

let srv = micro(pictures)

let url = await listen(srv)

let body = await request({ uri: `${url}/${id}`, json: true })

t.deepEqual(body, { id })

})

• A micro en este caso le pasamos nuestra funcion que exportamos de nuestro microservicio y luego en la url a listen le pasamos el server que acabamos de crear para que nos entregue el puerto y la url en la que esta corriendo.

Creacion de nuestro endpoint GET /picture

Vamos a implementar el endpoint de la petición de una imagen. No olvidemos que, nuestro API, no va a acceder directamente a la base de datos, si no que usara nuestro módulo de base de datos. Los fixtures nos ayudarán a tener datos similares a los esperados en nuestros test.

Para el entorno de prueba, no es necesario conectarse a nuestra base de datos real, por lo tanto, haremos uso de un stub gracias a la librería proxyrequire

Vamos a crear nuestro fixture que simulara los datos que nos tendra que entrega nuestro endpoint:

export default {

getImage () {

return {

id: 'a78c21ed-2f7b-407a-aea7-2d49ffe13a5c',

publicId: '569L4WkW4duuR3cVM83Q3G',

userId: 'platzigram',

src: 'http://platzigram.test/569L4WkW4duuR3cVM83Q3.jpg',

description: '#awesome',

tags: ['awesome'],

createdAt: new Date().toString()

}

}

}

Y ahora cambiamos nuestro test para que nos reciba el fixture que creamos y que compare el resultado con la imagen que creamos en el fixture. Este seria nuestro codigo:

import fixtures from './fixtures/index'

test('GET /:id', async t => {

let image = fixtures.getImage()

let srv = micro(pictures)

let url = await listen(srv)

let body = await request({ uri: `${url}/${image.publicId}`, json: true })

t.deepEqual(body, image)

})

Ahora vamos a ver como implementar estos test ya que no estamos jugando con la base de datos, veamos como lo hacemos. Entonces vamos a usar el modulo de platzigram-db, pero como nosotros no tenemos publicado este modulo, podemos hacerlo de la siguiente manera:

Nos posamos en la carpeta contenedora del modulo de platzigram-db y ejecutamos los siguientes comandos:

npm i

Este es para serciorarnos de que todas las dependencias esten instaladas y el modulo este 100% usable.

npm link .

Este nos sirve para enlazar nuestro modulo al entorno global de npm que tenemos instalado a nuestra maquina.

Ahora vamos a posarnos en nuestro modulo de platzigram-api y ejecutamos los siguientes comandos:

npm link platzigram-db

Este para que nuestro modulo de la api nos tenga contenido o como dependencia el modulo de la base de datos

Esta es una manera facil de enlazar modulos de forma local como para cuando se esta probando funciones y demas.

Vamos a usar **stubs** que son funciones creadas especificamente para hacer pruebas sobre una funcionalidad. Asi se le conoce en testing a estas funciones.

Vamos a crear nuestra clase de **Db** “falsa” en la que solo vamos a crear los metodos que nos retornaran las promesas resueltas en true y los resultados que tendria que devolver la clase de base de datos verdadera. Veamos el codigo:

'use strict'

import fixtures from '../fixtures/index'

export default class Db {

connect () {

return Promise.resolve(true)

}

disconnect () {

return Promise.resolve(true)

}

getImage (id) {

return Promise.resolve(fixture.getImage())

}

}

Ahora como vamos a decidir que vamos a usar el **stub** o la clase real de la base de datos. Para este caso vamos a usar un modulo que se llama **proxyquire**, entonces vamos a usar este para definir que vamos a usar un **stub** a la hora de implementar pruebas.

Esta es la documentacion de este, es una buena alternativa para usar **stubs** de la manera de common js <https://github.com/thlorenz/proxyquire>

En nuestro caso vamos a hacerlo de la forma antigua, asi quedaria nuestro codigo:

import Db from 'platzigram-db'

import config from './config'

import DbStub from './test/stub/db'

const env = process.env.NODE\_ENV || 'production'

let db = new Db(config.db)

if (env === 'test') {

db = new DbStub()

}

const hash = HttpHash()

hash.set('GET /:id', async function getPicture (req, res, params) {

let id = params.id

await db.connect()

let image = await db.getImage(id)

await db.disconnect()

send(res, 200, image)

})

• Usamos una variable de entorno mediante nodeJS y si esta no esta definida le pondremos por defecto el valor de  **production** para indicarle a nuestro codigo que vamos a ejecutar este como produccion y no como prueba.  
 • Luego evaluamos si esta variable esta en ‘test’ que use nuestro **stub** para implementar los test, pero si no seteamos este valor el defecto sera un codigo para produccion.  
 • Luego cuando ejecutemos nuestro codigo como “production” le vamos a pasar una configuracion a la base de datos para se adapte a cualquier sistema si es que migramos a otro en un futuro. Primero creamos un archivo llamado config.js y en este iremos poniendo toda la configuracion que queramos. Por el momento estara vacia:

export default {

db: {}

}

Ahora vamos a nuestro archivo **package.json** y vamos a agregar las siguientes lineas en los test para agregar el valor de test a nuestra variable de entorno que evaluamos en el codigo.

"test": "npm run lint && ava test/\*\*-test.js"

• Esto es para que solo testee los archivos que tengan en su definicion “-test.js”

Luego agregamos el siguiente:

"ava": {

"require": [

"babel-register"

],

"NODE\_ENV": "test"

}

• Este es para setear el valor de NODE\_ENV en test cuando se ejecute el comando de **ava**

Creacion de nuestro endpoint POST /picture

Vamos a implementar el endpoint para subir una imagen. No olvidemos que, nuestro API, no va a acceder directamente a la base de datos, si no que usara nuestro módulo de base de datos. Los fixtures nos ayudarán a tener datos similares a los esperados en nuestros test.

Vamos a definir la ruta POST para nuestra ruta de /picture veamos el codigo para nuestro test:

test('POST /', async t => {

let image = fixtures.getImage()

let url = t.context.url

let options = {

method: 'POST',

uri: url,

json: true,

body: {

description: image.description,

src: image.src,

userId: image.userId

},

resolveWithFullResponse: true

}

let response = await request(options)

t.is(response.statusCode, 201)

t.deepEqual(response.body, image)

})

• Como este request es un POST va a estar esperando que nosotros le enviemos la url, el metodo, el body que nosotros queremos enviar y ademas con **resolveWithFullResponse** garantizamos que la librería **request** nos devuelva todo el contenido del response para poder usar el **statusCode** en la comparacion.  
 • Creamos una instancia de request que es la que nos hara el request a nuestra ruta y le pasamos por parametro las opciones que definimos antes.  
 • Ahora como vamos a usar el server con micro para los test y para no tener que repetir codigo en cada uno de los test vamos a hacer uso de la propiedad de **ava** *beforeEach* para ejecutar el mismo microservicio para todas nuestras pruebas a las rutas. Este es el codigo para esto:

test.beforeEach(async t => {

let srv = micro(pictures)

t.context.url = await listen(srv)

})

Ahora vamos a implementar la ruta que vamos a probar en nuestro microservicio:

hash.set('POST /', async function postPicture (req, res, params) {

let image = await json(req)

await db.connect()

let created = await db.saveImage(image)

await db.disconnect()

send(res, 201, created)

})

• Con **HttpHash** no podemos acceder al body de un request normalmente, de esta manera como importamos modulos usando ES2015 vamos a exportar la funcion **json()** que nos extrae todo el contenido del body del request, luego implementamos la logica para nuestra base de datos u prueba según se decida. Este es el codigo:

import { send, json } from 'micro'

Creacion de nuestro endpoint POST /picture/:id/like

Vamos a implementar el endpoint para agregar los likes de una imagen, según el ID. No olvidemos que, nuestro API, no va a acceder directamente a la base de datos, si no que usara nuestro módulo de base de datos. Casi lo mismo que el ejercicio anterior.

Vamos como siempre a comenzar por los test:

test('POST /:id/like', async t => {

let image = fixtures.getImage()

let url = t.context.url

let options = {

method: 'POST',

uri: `${url}/${image.id}/like`,

json: true

}

let body = await request(options)

let imageNew = JSON.parse(JSON.stringify(image))

imageNew.liked = true

imageNew.like = 1

t.deepEqual(body, imageNew)

})

* Aquí incluimos una forma de clonar un objeto en JS mediante el parseo y el parseo de nuevo de un objeto JSON, esto no es muy optimo para cosas complejas pero para el test que va a ser una sola imagen nos sirve.

Ahora creamos el metodo en nuestro stub y el fixture para que sea consumido por la prueba:

likeImage (id) {

let image = fixtures.getImage()

image.liked = true

image.like = 1

return Promise.resolve(image)

}

Ahora veamos la implementacion de nuestra ruta en hash:

hash.set('POST /:id/like', async function likePicture (req, res, params) {

let id = params.id

await db.connect()

let image = await db.likeImage(id)

await db.disconnect()

send(res, 200, image)

})

* Nunca olvidar resolver las promesas con la palabra **await** o si no nos retornara un objeto mas complejo y nos arrojara un error el test.

Creacion de nuestro endpoint GET /picture/list

Vamos a implementar el endpoint para obtener el listado de todas nuestras imágenes. No olvidemos que, nuestro API, no va a acceder directamente a la base de datos, si no que usara nuestro módulo de base de datos. Casi lo mismo que el ejercicio anterior.

Vamos a definir esta ruta primero que la ruta del GET /:id ya que se genera una colision y lo que queremos es que se mantenga la presedencia de las rutas, entonces para que no entre primero en la del id y haga lo que quiere en esta la definimos de primera.

Este problema ocurre tambien en el enrutar de Express o del ecosistema de Node.

Primero como siempre vamos a implementar nuestro test:

test('GET /list', async t => {

let image = fixtures.getImages()

let url = t.context.url

let options = {

method: 'GET',

uri: `${url}/list`,

json: true

}

let body = await request(options)

t.deepEqual(body, image)

})

Esta vez vamos a crear un fixture que nos devuelva simplemente 3 imágenes para probar que esta listando las imágenes, entonces veamos el codigo del fixture:

getImages () {

return [

this.getImage(),

this.getImage(),

this.getImage()

]

}

Y ahora definamos la implementacion para nuestra ruta de Listar imágenes, veamos el codigo:

hash.set('GET /list', async function list (req, res, params) {

await db.connect()

let images = await db.getImages()

await db.disconnect()

send(res, 200, images)

})

* Esta ruta tiene que estar por encima de todas ya que como definimos una ruta GET con parametro “GET /:id” y si se define esta por encima de todas va a tomar cualquier ruta GET / como match entonces entrara a esta y retornara los datos que no estamos esperando. Para solucionar eso lo unico que hacemos es definir nuestra ruta de listar imágenes primero y luego la del :ID.
* Luego hacemos el procedimiento normal de conectarnos a la base de datos o al stub y luego resolvemos las promesas que nos entrega nuestro modulo de base de datos o nuestro stub de pruebas.

Creacion de nuestro endpoint GET/picture/tag

Vamos a implementar el endpoint para obtener las imágenes, según un tag determinado. No olvidemos que, nuestro API, no va a acceder directamente a la base de datos, si no que usara nuestro módulo de base de datos. Casi lo mismo que el ejercicio anterior.

Recordar que siempre que corremos un test estamos creando un servidor para correr este microservicio. Asi para todos los test.

Como siempre vamos a crear nuestro test:

test('GET /tag/:tag', async t => {

let images = fixtures.getImagesByTag()

let url = t.context.url

let options = {

method: 'GET',

uri: `${url}/tag/awesome`,

json: true

}

let body = await request(options)

t.deepEqual(body, images)

})

Y luego vamos a crear la implementacion de este test:

hash.set('GET /tag/:tag', async function byTag (req, res, params) {

let tag = params.tag

await db.connect()

let images = await db.getImagesByTag(tag)

await db.disconnect()

send(res, 200, images)

})

Pero antes de terminar esta implementacion de esta ruta tenemos que crear tambien nuestro metodo en el stub que nos devuelva las imágenes para la prueba:

getImagesByTag (tag) {

return Promise.resolve(fixtures.getImagesByTag())

}

Y lo mismo en el fixture:

getImagesByTag () {

return [

this.getImage(),

this.getImage()

]

}

Creacion de nuestro endpoint POST /user

Ahora vamos a crear nuestro microservicio para las rutas de los users.

Entonces creamos un archivo aparte para nuestro micro servicio y para nuestro archivo de test para nuesto microservicio.

Entonces como siempre empezamos por los test de nuestra ruta de micro servicio:

test('POST /', async t => {

let user = fixtures.getUser()

let url = t.context.url

let options = {

method: 'POST',

uri: url,

json: true,

body: {

name: user.name,

username: user.username,

password: user.password,

email: user.email

},

resolveWithFullResponse: true

}

let response = await request(options)

delete user.email

delete user.password

t.is(response.statusCode, 201)

t.deepEqual(response.body, user)

})

* Vamos a eleminar del objeto las propiedades del email y el password gracias al operador **delete** que nos permite eliminar propiedades de objetos o variables que son configurables.
* Aquí como es nuestro registro necesitamos guardar la contraseña en texto plano, despues es que vamos a encriptarla.

Ahora vamos a crear nuestros metodos en el fixture y el stub de base de datos:

getUser () {

return {

id: 'af374136-4f84-4ffa-8c72-499eb2e4e52d',

name: 'Steven Anaya',

username: 'stvanaya',

email: 's@platzi.test',

password: 'platzi',

createdAt: new Date().toString()

}

}

Y luego el de la base de datos:

saveUser (user) {

return Promise.resolve(fixtures.getUser())

}

Ahora veamos como es la implementacion de nuestra ruta en el microservicio de los usuarios:

hash.set('POST /', async function saveUser (req, res, params) {

let user = await json(req)

await db.connect()

let created = await db.saveUser(user)

await db.disconnect()

delete created.email

delete created.password

send(res, 201, created)

})

* Aquí vamos a hacer todo como lo hemos venido haciendo pero aca cuando vayamos a devolver el objeto resultante de la respuesta le tenemos que eliminar los campos sensilbles o que nosotros creamos que son privados como lo son el email y el password ya que no queremos que estos vayan al frontend como datos para mostrar o como datos que pueden ser accedidos, entonces hacemos uso de nuevo del operador **delete** para elimarle propiedades al objeto que tiene que ser configurable, si no lo es este nos devolvera una excepcion o un false.
* Luego devolvemos la respuesta y el statusCode que es 201 que significa completado exitosamente y que se creo el objeto.

Creacion de nuestro endpoint GET /user-name

Como hemos venido hecho vamos a empezar por nuestros test primero, asi que veamos el codigo de nuestro test:

test('GET /:username', async t => {

let user = fixtures.getUser()

let url = t.context.url

let options = {

method: 'GET',

uri: `${url}/${user.username}`,

json: true

}

let body = await request(options)

delete user.email

delete user.password

t.deepEqual(body, user)

})

* Aplicamos la misma tecnica de seguridad para eliminar los datos sensibles del objeto del usuario, luego comprobamos que sean iguales el que nos entrega el stub con el que nos devuelve el request.

Ahora necesitamos implementar nuestro metodo dentro del stub para obtener el usuario para la prueba.

getUser (username) {

return Promise.resolve(fixtures.getUser())

}

Ya con esto implementado vamos a crear nuestra ruta para obtener nuestra ruta para obtener un usuario:

hash.set('GET /:username', async function getUser (req, res, params) {

let username = params.username

await db.connect()

let user = await db.getUser(username)

await db.disconnect()

delete user.email

delete user.password

send(res, 200, user)

})

* Usamos basicamente lo que hemos estado usando en las anteriores rutas, resolvemos las promesas de las conexiones a la base de datos, nos desconectamos y devuelta eliminamos los datos sensibles del usuario por temas de seguridad. Como ultimo paso tendriamos el envio de la respuesta del request hecho.

Ahora vamos a crear nuestra ruta para poder subir una foto, pero nosotros queremos que solo un usuario registrado y logeado pueda subir una foto, entonces vamos a ver como podemos restringir el acceso a nuestra API REST con las siguientes herramientas.

Creacion de Tokens para nuestro API usando JSON Web Tokens

Ya tenemos nuestras rutas definidas pero nos hacen falta 1, y para implementar esta necesitamos asegurarla para que solo pueda accederse mediante un Token, entonces hay un estandar para esto llamado **JSON Web Tokens** el cual vamos a usar cuando entremos en el proceso de autenticacion y que nuestra API requerira este Token que nos entrega para poder realizar peticiones HTTP.

Con este estandar garantizamos que un “Proveedor” nos verifique que si estamos obteniendo la informacion correcta, puedo enviar informacion en el Token y podemos asegurarnos de que el Token se destruya o que sea invalidado cuando terceros intercepten la informacion para sus beneficios.

Las dos partes que van a hacer parte de esto, ejemplo, nuestra aplicación y el proveedor de los Tokens necesitan tener un *Secret Key* que deben conocer los dos o utilizar un mecanismo de encripcion de **llave publica/llave privada** donde yo le doy la llave publica al que va a consumir el Token y luego yo lo firmo con mi llave privada para de esta forma autenticar esto.

En el caso de esta aplicación vamos a hacer uso del **JWT** con la llave secreta compartida. Donde nuestra aplicación va a conocer y la API va a conocer y la hora de realizar la peticion vamos a validar con estas dos llaves la informacion para poder confirmar que esta es correcta.

**Token**: Es un String extenso donde tenemos informacion encriptada.

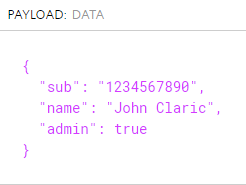
Aca podemos ver la documentacion de los **JWT**: <https://jwt.io/>

Un **JWT** consta basicamente de 3 partes:

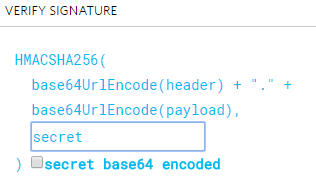
1. **Header**: Esta contiene la informacion sobre el algoritmo que va a utilizar para encriptar la informacion.



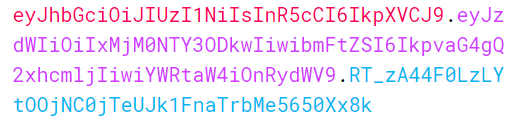
1. **Payload**: Esta con tiene todos los datos que nosotros queremos compartir, tenemos que tener en cuenta que esta informacion la podra ver cualquier persona, entonces no tenemos que usar informacion sensible en esta por seguridad.



1. **Verify Signature (Firma de verificacion)**: Esta es la parte de seguridad de esto, esta hace la verificacion de que la encriptacion del header + el payload y la palabra **Secreta** que hayamos elegido sean correctas, todo esto lo hace usando el algoritmo SHA256 para codificar este. El punto fuerte de esto es que cuando alguien se quiere pasar de listo modificando el Payload para ingresar con permisos o para modificar el **JWT** lo que va hacer es verificar que se modifico el valor de la encriptacion de este Payload y entonces no va a pasar la validacion cuando este verifique el Token con el que tiene el sistema.



Esta es la base de como se ve el token:



La unica forma de que se pueda generar un Token valido es conociendo el **Secret**, de otra forma esto siempre nos arrojara un error de que este Token no es valido, pero este error lo vamos a generar nosotros en la aplicación.

En este caso nosotros vamos a instalar la dependencia para **JWT** para Node patrocinada por la compañía de auth0 que fue la que patrocino todo el desarrollo de esta tecnologia de **JWT**.

npm install jsonwebtoken --save

Ahora vamos a crear un archivo Utils para crear y verificar los Tokens que vayamos a usar.

El estandar recomienda pasar el Token en el header del request HTTP en la parte de authorization. Podemos tambien pasarlo por parametro de la ruta, o por medio de un POST, pero la recomendación de este es con el header.

Vamos a instalar otro modulo para extraer el Token del header del request:

npm install token-extractor --save

Ahora vamos a crear nuestro modulo de Util para crear los metodos que necesitamos para manejar los **JWT**, tenemos que hacer basicamente los metodos para firmar un Token, para verificar un Token y otro para extraer el Token del header del request que nos llega.

'use strict'

import jwt from 'jsonwebtoken'

import bearer from 'token-extractor'

export default {

async signToken (payload, secret, options) {

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

jwt.sign(payload, secret, options, (err, token) => {

if (err) return reject(err)

resolve(token)

})

})

},

* Creamos una funcion asincrona para manejar todo con el **await**, pero **JWT** las funciones asincrona con callbacks y entonces en el caso de que nosotros queremos manejar una promesa al retornar, vamos a usar la forma completa de escribir una promesa.
* La funcion nos debe recibir el Payload(Data que queremos pasar), el Secret(La clave secreta) y las opciones que queramos modificar, por ejemplo podemos modificar el algoritmos de encriptacion o si el Token tiene algun tipo de vigencia.
* Luego hacemos uso del metodo de la librería de **JWT** sign() que nos recibe el Payload, el Secret, las opciones y luego un callback que nos recibe el error o el token.
* Por ultimo cuando ocurra el callback verificaremos si hubo un error o si fue exitosa la resolvemos y le pasamos el Token que nos devuelve **JWT.**

El siguiente metodo es para verificar los Tokens, veamos el codigo:

async verifyToken (token, secret, options) {

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

jwt.verify(token, secret, options, (err, decoded) => {

if (err) return reject(err)

resolve(decoded)

})

})

},

* Seguimos casi la misma estructura que la del metodo que creamos anteriormente. La diferencia es que este metodo no nos recibe el Payload, nos recibe el Token que queremos verificar.
* Luego vamos a usar el metodo de **JWT** verify() que nos recibe por parametro el token, el secret y las opciones ademas del callback que se va a ejecutar donde vamos a recibir el error y el Token decodificado con la data que queremos ver.
* Por ultimo manejamos los errores haciendo reject de la promesa o resolviendola pasando la Payload que decodificamos.

El ultimo metodo es para extraer el Token del header del request, para eso hacemos uso de la librería de **token-extractor** que instalamos antes. Este es el codigo del metodo:

async extractToken (req) {

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

bearer(req, (err, token) => {

if (err) return reject(err)

resolve(token)

})

})

}

* En este recibimos por parametro el request HTTP que recibimos.
* Luego hacemos uso del modulo que instalamos para extraer el Token del header, este nos recibe el request y un callback para manejar el error o el token.
* Por ultimo manejamos los errores con reject o el resultado exitoso de la promesa con resolve pasandole el token que extraimos del header del request.

Limitando nuestros request esperando recibir un Token

Como siempre vamos primero a realizar nuestros test. Que por cierto son una maricada toda marica, por que fallan por la creacion de la fecha que tienen de diferencia 1 segundo, pero de resto esta todo bien.

Ahora vamos a ver el codigo de los test:

import utils from '../lib/utils'

import config from '../config'

test('no token POST /', async t => {

let image = fixtures.getImage()

let url = t.context.url

let options = {

method: 'POST',

uri: url,

json: true,

body: {

description: image.description,

src: image.src,

userId: image.userId

},

resolveWithFullResponse: true

}

await t.throws(request(options), /invalid token/)

})

* Aca creamos un test que falle cuando no le pasamos un Token al al header del request. Entonces lo que vamos a hacer es que no realizamos las aserciones si no que vamos a lanzar una excepcion con el test y mediante una expresion regular nos busque la palabra del error marcada. Aun no hacemos el request al microservicio.

test('secure POST /', async t => {

let image = fixtures.getImage()

let url = t.context.url

let token = await utils.signToken({ userId: image.userId }, config.secret)

let options = {

method: 'POST',

uri: url,

json: true,

body: {

description: image.description,

src: image.src,

userId: image.userId

},

headers: {

'Authorization': `Bearer ${token}`

},

resolveWithFullResponse: true

}

let response = await request(options)

t.is(response.statusCode, 201)

t.deepEqual(response.body, image)

})

* Aca creamos el test para el test donde si le pasamos el Token, entonces hacemos uso del utils que creamos para firmar un Token, le pasamos por parametro el Payload y luego el secret.
* Luego en las opciones de nuestro request creamos el apartado para el header del request y ahí creamos la clave Authorization con la palabra Bearer y el token que generamos.
* Luego hacemos el request y hacemos las aserciiones.
* Tenemos que crear en nuestro archivo de configuracion la variable de entorno para el secret, debemos recordar no guardar esto en ninguna parte del codigo, ni mucho menos usar valor por defecto para setear estas llaves secretas, veamos el codigo de este archivo config:

export default {

db: {},

secret: process.env.PLATZIGRAM\_SECRET || 'platzi' // Ojo no usar valores default en casos de produccion reales. Esto es a modo de ejemplo para la clase.

}

Ahora vamos a realizar la implementacion de la validacion de los tokens en las rutas de nuestro microservicio.

hash.set('POST /', async function postPicture (req, res, params) {

let image = await json(req)

try {

let token = await utils.extractToken(req)

await utils.verifyToken(token, config.secret)

} catch (err) {

return send(res, 401, { error: 'invalid token' })

}

await db.connect()

let created = await db.saveImage(image)

await db.disconnect()

send(res, 201, created)

})

* Lo que hacemos es encerrar en un Try/Catch la validacion de este y que si no es el token valido nos retorne el codigo 401 de error de authorizacion.
* Luego dentro verificamos con nuestra utilidad el token y si esta todo correcto nos autenticara y nos conectaremos a la base de datos para guardar la imagen.

Ahora vamos a crear el test para cuando alguien intercepte nuestro Token y le modifique el Payload. Veamos el codigo de este test:

test('hacky token POST /', async t => {

let image = fixtures.getImage()

let url = t.context.url

let token = await utils.signToken({ userId: 'hacky' }, config.secret)

let options = {

method: 'POST',

uri: url,

json: true,

body: {

description: image.description,

src: image.src,

userId: image.userId

},

headers: {

'Authorization': `Bearer ${token}`

},

resolveWithFullResponse: true

}

await t.throws(request(options), /invalid token/)

})

* Modificamos el userID con otro nombre en el Payload, luego hacemos basicamente lo mismo que el test anterior.
* La diferencia es que cuando corramos este test tal cual como esta este no va a pasar ya que espera que nuestra ruta le devuelva una excepcion al validar el token, pero como no estamos comparando el Token real del usuario que esta tratando de subir la imagen con el que nos esta entregando le request que es malicioso.

Veamos la logica para implementar esto:

hash.set('POST /', async function postPicture (req, res, params) {

let image = await json(req)

try {

let token = await utils.extractToken(req)

let encoded = await utils.verifyToken(token, config.secret)

if (encoded && encoded.userId !== image.userId) {

return send(res, 401, {error: 'invalid token'})

}

} catch (err) {

return send(res, 401, { error: 'invalid token' })

}

await db.connect()

let created = await db.saveImage(image)

await db.disconnect()

send(res, 201, created)

})

* Ahora vamos a hacer una verificacion de que si el token que obtenemos de nuestra funcion verifyToken() es igual al de la imagen que obtenemos, entonces que si modifican el Token no podran entrar ya que no los dejara entrar y nos mostrara este error.
* Ahora veamos un refactor del error ya que lo hacemos dos veces:

try {

let token = await utils.extractToken(req)

let encoded = await utils.verifyToken(token, config.secret)

if (encoded && encoded.userId !== image.userId) {

throw new Error('invalid token')

}

} catch (err) {

return send(res, 401, { error: 'invalid token' })

}

Ahora veamos como implementar la generacion del token cuando el usuarios se autentifique o vaya a hacer sign in en la aplicación y que este genere el token y luego se utilice para los demas request que lo necesiten.

Test de autenticacion de usuario

Ahora vamos a crear otro microservicio para la implementacoin de la autenticacion de un usuario, entonces vamos a crear un nuevo archivo para el microservicio y para los test.

Vamos a crear primero el test, veamos el codigo:

test('success auth POST /', async t => {

let user = fixtures.getUser()

let url = t.context.url

let options = {

method: 'POST',

uri: url,

body: {

username: user.username,

password: user.password

},

json: true

}

let token = await request(options)

let decoded = await utils.verifyToken(token, config.secret)

t.is(decoded.username, user.username)

})

* Mandamos en el body las credenciales que obtenemos del fixture.
* Luego verificamos el token que nos entrega la respuesta de la peticion y lo comparamos con el Payload que tiene con las credenciales que obtenemos de nuestro fixture de getUser.

Ahora veamos la implementacion de nuestro microservicio para la authenticacion de los usuarios. Este es el codigo:

hash.set('POST /', async function authenticate (req, res, params) {

let credentials = await json(req)

await db.connect()

let auth = await db.authenticate(credentials.username, credentials.password)

if (!auth) {

return send(res, 401, { error: 'invalid credentials' })

}

let token = await utils.signToken({

username: credentials.username

}, config.secret)

send(res, 200, token)

})

* Hacemos el llamado del metodo de autenticar de nuestra clase o stub según decidamos si estamos probando o no. Le pasamos el username y la password el usuario que se esta logeando.
* Luego comparamos que si nuestro metodo authenticate nos retorna un true o false, si nos retorna false este es que fallo la autenticacion del usuario y le retorna el error 401 de credenciales invalidas.
* Si es true el resultado generamos el Token y le pasamos el Payload que nosotros vamos a querer poner que en este caso es el nombre de usuario del usuario que se logeo. Por ultimo le pasamos el secret que tenemos definido.
* Devolvemos la respuesta con un 200 y tambien devolvemos el token para despues ser consumido.

Aquí terminamos nuestro modulo de API REST en el cual implementamos los test y los microservicios para cada uno de estas funcionalidades. Una gran ventaja de usar microservicios es que si alguno de estos tiene recarga o algun issue de performance lo podemos escalar individualmente sin afectar a los demas.

Tenemos una aplicación modular y con microservicios que nos van a controlar estos request y la respuesta que no entrega la base de datos para luego mostrarlos en nuestro Frontend. El benificio es grande ya que tenemos toda nuestra aplicación dividida en modulos independientes que si alguno falla no va a tirar toda nuestra aplicación.

Creacion de nuestro cliente

Ahora vamos a crear nuestro cliente, aca vamos a usar un “wrapper” que nos permita controlar los request que nos llegan a la API desde nuestro cliente. Veamos como hacerlo.

Este es como una librería que nos permitirar hacer una abstraccion de los request que nos van llegando, este modulo se llamara PLATZIGRAM\_CLIENT. Este nos permite entonces no requerir una librería para hacer http request si no crear una que nos retorne los datos que nosotros queremos.

Crearemos el modulo que conectar nuestro Frontend con nuestro API para consumir los datos.

Vamos a crear nuestro modulo con npm init y vamos a instalar las dependencias de desarrollo qu hemos estado usando:

npm install --save-dev ava babel-eslint standard uuid-base62

Ahora vamos a crear un archivo para nuestros fixtures, igual como hemos usado para los demas modulos. Veamos el codigo:

'use strict'

const uuid = require('uuid-base62')

const fixtures = {

getImage () {

let id = uuid.uuid()

return {

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

tags: [ 'awesome', 'tags', 'platzi'],

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

likes: 0,

liked: false,

userId: uuid.uuid(),

publicId: uuid.encode(id),

id: id,

createdAt: new Date().toString()

}

},

getImages (n) {

let images = []

while (n-- > 0) {

images.push(this.getImage())

}

return images

},

getUser () {

return {

id: uuid.uuid(),

name: 'A ramdom user',

username: 'Stvanaya',

createdAt: new Date().toString()

}

}

}

Y ahora vamos a configurar standard y el comando para este:

"scripts": {

"lint": "standard",

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

},

"standard": {

"parser": "babel-eslint"

},

Ahora vamos a crear nuestros test para este modulo:

'use strict'

const test = require('ava')

const platzigram = require('../')

// const fixtures = require('./fixtures')

test('client', t => {

const client = platzigram.createClient()

t.is(typeof client.getPicture, 'function')

t.is(typeof client.savePicture, 'function')

t.is(typeof client.likePicture, 'function')

t.is(typeof client.listPictures, 'function')

t.is(typeof client.listPicturesByTag, 'function')

t.is(typeof client.saveUser, 'function')

t.is(typeof client.getUser, 'function')

t.is(typeof client.auth, 'function')

})

Vamos a requerir una clase que nos entregue basicamente los metodos que se implementan en la API, pero los vamos a contener en una librería que nos servira como “WRAPPER”

Y luego vamos a crear la clase que vamos a estar exportando:

'use strict'

class Client {

getPicture () {}

savePicture () {}

likePicture () {}

listPictures () {}

listPicturesByTag () {}

saveUser () {}

getUser () {}

auth () {}

}

module.exports = Client

Todo hasta ahora vacio para que almenos pasen los test. Luego vamos a implementar todos estos metodos y vamos a testearlos.

Implementando el perfil de usuario

Para crear los test de este modulo vamos a usar como en el modulo de DB que o en el de API que usamos un Stub de nuestra base de datos falsa para probar que nuestro API hacia el llamado a nuestra base de datos correctamente. En este caso vamos a hacer mocking de request HTTPs, o sea vamos a crear un servidor falso para definir las rutas las cuales vamos a usar para testear nuestro modulo wrapper de PlatzigramClient.

Vamos a asegurar que cuando se haga el request HTTP y nuestro wrapper API llame los metodos y rutas correctas en nuestra API. Para esto vamos a usar interceptacion de la funcionalidad HTTP request del modulo de Node.

Instalaremos una librería para hacer mocking llamada nock. Esta es la documentacion:

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

La implementacion basica de esta librería es esta:

var nock **=** require('nock');

var couchdb **=** nock('http://myapp.iriscouch.com')

                .get('/users/1')

                .reply(200, {

                  \_id**:** '123ABC',

                  \_rev**:** '946B7D1C',

                  username**:** 'pgte',

                  email**:** 'pedro.teixeira@gmail.com'

                 });

* Donde le pasamos la url donde queremos hacer el HTTP.
* El metodo del request.
* Y luego la respuesta que queremos entregar en el test.

Ahora vamos a instalarlo como dependencia de desarrollo para nuestro test:

npm install --save-dev nock

En este caso nosotros tenemos 3 microservicios, 1 para las imágenes, 1 para los usuarios y otro para la autenticacion. Esto significa que vamos a tener corriendo 3 servidores HTTP con diferente puerto cada uno, entonces tendremos que definir programaticamente que ruta es la que estaremos solicitando en el request y ademas en produccion usaremos NGNX que nos servira como una capa que envuelve nuestros servicios entregando 1 solo al cliente para hacer mas facil el despliegue.

Estos 3 servidores serian nuestros ENDPOINTS de nuestra API REST que esta compuesta por 3 microservicios.

Veamos como vamos a crear los test para esto:

Primero vamos a crear un beforeEach para nuestros test con el context de nuestro cliente creado, este es el codigo:

test.beforeEach(t => {

t.context.client = platzigram.createClient(options)

})

Ahora crearemos las opciones que pasaremos a cada uno de estos test. Veamos el codigo:

let options = {

endpoints: {

pictures: 'http://platzigram.test/picture',

users: 'https://platzigram.test/user',

auth: 'http://platzigram.test/auth'

}

}

* Estos endpoints son las rutas que usariamos o que configuraremos cuando implementemos con NGNX, ejemplo si queremos acceder a nuestro Endpoint de picture la ruta seria la **pictures** pero seguido del / y la ruta que le asignamos en nuestro microservicio.

Con esto ya creado vamos a empezar a definir nuestros test. Empezaremos por el getPicture.

Vamos a instalar las siguientes dependencias asi como hicimos para manejar las respuestas hibridas que si nos pasan un callback lo ejecutamos como una funcion asincrona y si no lo devolvemos como una promesa.

npm install --save request request-promise bluebird

test('getPicture', async t => {

const client = t.context.client

let image = fixtures.getImage()

nock(options.endpoints.pictures)

.get(`/${image.publicId}`)

.reply(200, image)

let result = await client.getPicture(image.publicId)

t.deepEqual(image, result)

})

Ahora vamos a implementarlo en nuestro cliente:

Primero vamos a definir nuestra configuracion de la clase en el constructor, veamos el codigo:

constructor (options) {

this.options = options || {

endpoint: {

pictures: 'http://api.platzigram.com/picture',

users: 'http://api.platzigram.com/user',

auth: 'http://api.platzigram.com/auth'

}

}

}

* Entonces si no nos pasan opciones, por defecto vamos a usar las rutas de nuestros endpoint que vamos a configurar cuando hagamos deploy de nuestra aplicación. Ahora veamos la implementacion del metodo de getPictures()

Veamos el codigo:

getPicture (id, callback) {

let opts = {

method: 'GET',

uri: `${this.options.endpoint.pictures}/${id}`,

json: true

}

return Promise.resolve(request(opts)).asCallback(callback)

}

* Vamos a recibir por parametro el id de la imagen que se va a devolver y un callback opcional por si se quiere ejecutar como una funcion asincrona normal.
* Luego vamos a crear las opciones para pasarle a nuestro request.
  + En la uri le pasamos la ruta del endpoint que definimos junto al :id que espera la ruta que definimos.
* Por ultimo vamos a retornar la promesa resuelta y resolvemos el request, si pasamos un callback asincronico vamos a ejecutarlo como una funcion asincrona normal.

Carga de nuestras imágenes desde nuestro cliente

Vamos a crear nuestro test para el metodo de savePicture(), como esta ruta la definimos en nuestro microservicio como segura y que necesitara un Token para poder ser usada entonces le vamos a pasar un token ficticio.

Este es el codigo para nuestro test:

test('savePicture', async t => {

const client = t.context.client

let token = 'xxx-xxx-xxx'

let image = fixtures.getImage()

let newImage = {

src: image.src,

description: image.description

}

nock(options.endpoints.pictures, {

reqheaders: {

'Authorization': `Bearer ${token}`

}

})

.post('/', newImage)

.reply(201, image)

let result = await client.savePicture(newImage, token)

t.deepEqual(image, result)

})

* Vamos generar un token random. Obtenemos una imagen de los fixtures y creamos un objeto que va a contener una nueva imagen pero solo vamos a pasarle los campos del src y la descripcion, los demas campos no nos interesan ahora.
* Luego vamos a usar el mismo endpoint pero esta vez vamos a hacer no una peticion tipo GET si no una peticion tipo POST y le pasamos la imagen que queremos guardar.
  + Luego respondemos con el codigo que definimos y con la imagen creada.
  + Luego resolvemos resolvemos el metodo del client pasandole la nueva imagen y el token para acceder a la base de datos y a la API.
  + Tambien tenemos que decirle a nock que tenemos que usar un header en el request y creamos el objeto con la autorizacion y el header.

Ahora veamos la implementacion de nuestro metodo de savePicture(). Este es el codigo:

savePicture (image, token, callback) {

let opts = {

method: 'POST',

uri: `${this.options.endpoints.pictures}/`,

body: image,

headers: {

'Authorization': `Bearer ${token}`

},

json: true

}

return Promise.resolve(request(opts)).asCallback(callback)

}

* Primero le pasamos la imagen por parametro el token y el callback opcional.
* Luego en el body le pasamos la imagen a guardar y un header para pasar la autorizacion con el Token.
* Luego retornamos la promesa o el callback según se decida.

Ahora sigamos con nuestro metodo que seria likePicture(), veamos como creamos el test para este metodo:

test('likePicture', async t => {

const client = t.context.client

let image = fixtures.getImage()

image.liked = true

image.like++

nock(options.endpoints.pictures)

.post(`/${image.publicId}/like`)

.reply(200, image)

let result = await client.likePicture(image.publicId)

t.deepEqual(image, result)

})

* Tenemos la misma estructura del test de getPicture pero vamos a cambiar solo los campos de liked y like.
* Luego en nock hacemos un request de tipo post con la ruta y luego respondemos con la imagen.

Ahora veamos la implementacion del metodo, este es el codigo:

likePicture (id, callback) {

let opts = {

method: 'POST',

uri: `${this.options.endpoints.pictures}/${id}/like`,

json: true

}

return Promise.resolve(request(opts)).asCallback(callback)

}

* Creamos la ruta POST y la ruta que habiamos definido para nuestro microservicio y la ruta que tenemos en el nock de nuestro test.
* Luego retornamos la misma linea de codigo que la de los demas.

Ahora vamos a implementar los test para el metodo de listPictures(), como siempre vamos a crear nuestro test siempre siguiente el TDD, creo primero mi test con los contratos que quiero seguir y demas funcionalidad y luego si la implemento en el codigo fuente o principal.

Aquí esta el codigo del test:

test('listPicture', async t => {

const client = t.context.client

let images = fixtures.getImages(3)

nock(options.endpoints.pictures)

.get('/list')

.reply(200, images)

let result = await client.listPictures()

t.deepEqual(images, result)

})

* Creamos una arreglo de imágenes y retornamos este en el nock.
* Luego resolvemos nuestro metodo de listPictures y hacemos la asercion de nuestro resultado.

Aquí esta el codigo de la implementacion de este test:

listPictures (callback) {

let opts = {

method: 'GET',

uri: `${this.options.endpoints.pictures}/list`,

json: true

}

return Promise.resolve(request(opts)).asCallback(callback)

}

* Un test muy basico que no necesita explicacion.

Ahora vamos a implementar el test para nuestro metodo de listPicturesByTag(), este es el codigo:

test('listPictureByTag', async t => {

const client = t.context.client

let images = fixtures.getImages(3)

let tag = 'platzi'

nock(options.endpoints.pictures)

.get(`/tag/${tag}`)

.reply(200, images)

let result = await client.listPicturesByTag(tag)

t.deepEqual(images, result)

})

* Creamos un tag por el que queremos buscar y luego resolvemos nuestro metodo de la librería.

Ahora implementemos el test en la librería:

listPicturesByTag (tag, callback) {

let opts = {

method: 'GET',

uri: `${this.options.endpoints.pictures}/tag/${tag}`,

json: true

}

return Promise.resolve(request(opts)).asCallback(callback)

}

* En esta implementacion lo unico que cambia basicamente es la url al que queremos hacer el request.

Ahora vamos a hacer los metodos del usuarios y para la autenticacion.

Como siempre hemos venido haciendo primero creamos el test:

test('saveUser', async t => {

const client = t.context.client

let user = fixtures.getUser()

let newUser = {

username: user.username,

name: user.name,

email: 'platzi@hotmail.com',

password: 'platzi'

}

nock(options.endpoints.users)

.post('/', newUser)

.reply(201, user)

let result = await client.saveUser(newUser)

t.deepEqual(user, result)

})

* Creamos un usuario y le seteamos los campos que queremos cambiar.
* Luego en el nock configuramos la ruta para el endpoint de usuarios y hacemos el mismo procedimiento.
* Por ultimo resolvemos nuestro metodo saveUser y comparamos los resultado.

Ahora veamos la implementacion de este metodo de saveUser():

saveUser (user, callback) {

let opts = {

method: 'POST',

uri: `${this.options.endpoints.users}/`,

body: user,

json: true

}

return Promise.resolve(request(opts)).asCallback(callback)

}

* Creamos las opciones para el request y lo unico que cambia aquí es la ruta ya que nos pasamos al endpoint de los usuarios.

Ahora implementemos el test y el metodo para getUser():

test('getUser', async t => {

const client = t.context.client

let user = fixtures.getUser()

nock(options.endpoints.users)

.get(`/${user.username}`)

.reply(200, user)

let result = await client.getUser(user.username)

t.deepEqual(user, result)

})

* Basicamente lo mismo solo que cambia la ruta y el parametro que le pasamos a nuestro metodo a resolver.

Veamos la implementacion de este test:

getUser (username, callback) {

let opts = {

method: 'GET',

uri: `${this.options.endpoints.users}/${username}`,

json: true

}

return Promise.resolve(request(opts)).asCallback(callback)

}

* Es basicamente lo mismo solo que cambia la ruta.

AHORA FINALICEMOS ESTA CLASE CON EL ULTIMO METODO QUE SERIA EL DE AUTENTICAR USUARIOS.

Como siempre (Que repetitivo es esto de los TDD) vamos a realizar primero nuestro test:

test('auth', async t => {

const client = t.context.client

let credentials = {

username: 'stvanaya',

password: 'platzi'

}

let token = 'xxx-xxx-xxx'

nock(options.endpoints.auth)

.post('/', credentials)

.reply(200, token)

let result = await client.auth(credentials.username, credentials.password)

t.deepEqual(token, result)

})

* Para este caso vamos a pasar las credenciales de autenticacion, como lo es el password y el nombre de usuario.
* Leugo creamos un token que esperamos que sea la respuesta del nock.
* Luego resolvemos el metodo de la implementacion pasandole las credenciales que habiamos definido antes.

Ahora veamos la implementacion de este metodo:

auth (username, password, callback) {

let opts = {

method: 'POST',

uri: `${this.options.endpoints.auth}/`,

body: {

username,

password

},

json: true

}

return Promise.resolve(request(opts)).asCallback(callback)

}

}

* Recibimos por parametro el callback opcional y las credenciales del usuario.
* Creamos las opciones, y cambiamos la ruta para el endpoint de autenticacion.
* Enviamos en el body las credenciales del usuario y luego resolvemos el request.

Esto seria todo para este modulo, ahora veamos la implementacion para usar esta librería desde el frontend para poder autenticar usuarios y listar las imágenes.

Como implementar autenticacion de usuarios con Passport

Vamos a implementar middlewares en nuestro sistema para que validar que nuestros usuarios esten registrados o logeados cuando quieran subir una imagen. Ahora vamos a usar una librería llamada **Passport** que es muy popular en el ecosistema de NodeJS.

<http://www.passportjs.org/docs>

Configurando middlewares en express

Ahora vamos a trabajar sobre nuestro archivo de rutas del frontend que usamos con expreess en el curso definitivo de JS con sacha. Ahora vamos a instalar una serie de dependencias para poder autenticar un usuario con express:

Vamosa instalar algunos para el manejo de sesiones y la gestion de cookies.

npm install --save body-parser cookie-parser express-session

* Body-parser: es la encargada de hacer el parseo de la data que nos llegue a nuestra ruta de autenticacion y de registro de usuario.
* Cookie-parser: almacenar los datos de la autenticacion de la sesion en una cookie. Esto lo hace Passport automaticanmete por nosotros pero le debemos habilitar esta funcional a express.
* Express-session: me va a permitir guardar datos de la sesion en el lado del servidor para luego ser referenciados con una cookie.

Ahora veamos la implementacion de esto en el server.js:

var cookieParser = require('cookie-parser');

var bodyParser = require('body-parser');

var session = require('express-session');

app.set(bodyParser.json())

app.use(bodyParser.urlencoded({ extended: false }));

app.use(cookieParser());

app.use(session({

secret: config.secret,

resave: false,

saveUninitialized: false

}))

* Seteamos los middlewares, le decimos que nos parsee a json la data que nos llegue a estos middlewares.
* Luego que no sea tan extensa esta codificacion del json, y que maneje esto a cookies.
* Luego le decimos a express sesion , que obtega el secret key nuestro archivo de configuracion, que no guarde mas de una vez la sesion y que no almacene sesiones no inicializadas.

Ahora veamos el codigo del configjs:

'use strict'

const config = {

aws: {

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

secretKey: process.env.AWS\_SECRET\_KEY

},

secret: process.env.PLATZIGRAM\_SECRET || 'pl4tzi'

}

module.exports = config;

Ahora vamos a instalar Passport.js y las dependencias que necesitamos para correr estos..

npm install --save passport

* Este es el core del proyecto de passport, pero nosotros vamos a manejar dos formas de autenticar, una de forma local que es la convencional de Usuario y contraseña y la otra con Una red social, para esto tenemos que instalar los plugins para passport.

npm install passport-local passport-facebook --save

Ahora vamos a inicializar passport y que express nos use esta forma de autenticacion

var session = require('express-session');

var passport = require('passport');

app.use(passport.initialize());

app.use(passport.session());

Estrategia de autenticacion local (password/contraseña)

Aca vamos a validar nuestros campos que nos ingresen con los que tenemos almacenados en nuestra base de datos. Vamos a ver como podemos enlazar esta ruta de registrar usuario, para esto podemos hacer lo mismo que hicimos con npm y instalar como dependencia un modulo que nosotros hicimos sin tenerlo publicado en npm.

1. Primero nos posicionamos en la carpeta del platzigram-client y luego corremos el siguiente comando.

**npm link .**

1. Luego vamos a el modulo del platzigram donde tenemos el front-end y corremos el siguiente comando:

**Npm link platzigram-client**

1. Ahora ya podemos incluir este modulo en nuestro package.json como una dependencia de desarrollo.

Ahora vamos a requerir el modulo y lo vamos a instanciar.

var platzigram = require('platzigram-client')

var client = platzigram.createClient(config);

Luego lo que hacemos es crear un ruta post para poder guardar nuestros usuarios desde el frontend:

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

var user = req.body;

})

* Aca podemos hacer uso sin problema de nuestro body ya que con los middlewares que creamos el parser nos entrega ya limpito nuestra data del request.

Ahora como sabemos que tenemos 3 endpoints y que estos son lo accesibles llamando al wrapper o al platzigram-client, pero estos endpoints tienen que correr para que puedan funcionar, para esto vamos a tener que hacer una configuracion para que estos corran de manera local y asi poder hacer uso de estos endpoints.

'use strict'

const config = {

aws: {

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

secretKey: process.env.AWS\_SECRET\_KEY

},

secret: process.env.PLATZIGRAM\_SECRET || 'pl4tzi',

client: {

endpoints: {

pictures: 'http://api.platzigram.com/picture',

users: 'http://api.platzigram.com/user',

auth: 'http://api.platzigram.com/auth'

}

}

}

if (process.env.NODE\_ENV !== 'production') {

config.client.endpoints = {

pictures: 'http://localhost:5000',

users: 'http://localhost:5001',

auth: 'http://localhost:5002'

}

}

module.exports = config;

* Lo que hacemos es definir las rutas a las que tiene que ir a consultar, en primero definimos las rutas de produccion, las verdaderas, las que va a servir NGNX.
* Luego si nuestra variable de entorno no esta en produccion vamos a correr estos endpoints de forma local y con distinto puerto para cada uno de estos.

Ahora tenemos que seguir implementando nuestra ruta de POST para le signup:

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

var user = req.body;

client.saveUser(user, function (err, usr) {

if (err) return res.status(500).send(err.message);

res.redirect('/signin');

})

})

* Guardamos el usuario si esta todo bien y si no retornamos un mensaje.
* Luego redirigimos el usuario a la ruta del signin por que nosotros no vamos a logear de una vez al usuario. Luego el tiene que autenticarse.

Luego verificamos que en el template del form este bien especificado nuestra ruta y el metodo para hacer request.

Siguiente a esto tenemos que correr nuestro microservicios para poder consumirlos.

**Aca en este punto bota un error la librería de micro. Vamos a copiar tal cual lo que va a haciendo el profesor pero no va a funcionar al parecer por que hay una incompatibilidad de async con esta librería.**

Corremos los microservicios en ventanas diferentes de cmd con el siguiente comando.

**Micro –p 5000 picture.js**

Y asi con todos los endpoints que definimos.

Luego ya podemos hacer un registro y verificamos que en nuestra base de datos si se creo nuestro usuario.

Ahora veamos como traer los datos de la base de datos y compararlos para poder autenticar un usuario que se quiere registrar en nuestra plataforma.

Consulta, alta de usuarios y manejo de sesiones

Vamos a usar passportJS para manejar la sesiones, entonces lo que nos permite es definir los middlewares para cuando nosotros nos autentiquemos obtengamos una referencia del usuario que traemos de la base de datos. No guardamos todo el usuario por que sera contraproducente con la seguridad de la informacion.

Creamos un archivo nuevo en una carpeta nueva para no saturar nuestro **server.js**

Vamos a exportar tres funciones, veamos el codigo:

var LocalStrategy = require('passport-local').Strategy;

var platzigram = require('platzigram-client');

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

var client = platzigram.createClient(config.client);

exports.LocalStrategy = new LocalStrategy((username, password, done) => {

})

exports.serializeUser = function (user, done) {

}

exports.deserializeUser = function (user, done) {

}

* Requerimos el passport con la strategy de autenticacion.
* Creamos una instancia del cliente para realizar nuestros request.
* Primero creamos una funcion que recibe 3 parametros. 1 username, password y done que la vamos a llamar cuando nuestro procesos de autenticacion se complete.
* Como segundo paso creamos una funcion que nos va a serializar nuestro usuario. Tambien con un parametro done que vamos a llamar cuando este se complete.
* Por ultimo creamos el deserializador y le pasamos el usuario que tenemos guardado en la sesion.

Ahora vamos en nuestro archivo **server.js** a crear los middlewares para que passportJS sepa cual es la estrategia y los metodos para la serializacion y deserializacion de usuarios. Este es el codigo:

passport.use(auth.localStrategy);

passport.deserializeUser(auth.deserializeUser);

passport.serializeUser(auth.serializeUser);

Ahora vamos a aplicar la logica para el manejo de sesion de nuestro usuarios. Este es el codigo:

exports.LocalStrategy = new LocalStrategy((username, password, done) => {

client.auth(username, password, (err, token) => {

if (err) {

return done(null, false, { message: 'username and password not found'});

}

client.getUser(username, (err, user) => {

if (err) {

return done(null, full, { message: `an error ocurred: ${err.message}`})

}

user.token = token;

return done(null, user);

})

})

})

exports.serializeUser = function (user, done) {

done(null, {

username: user.username,

token: user.token

})

}

exports.deserializeUser = function (user, done) {

client.getUser(user.username, (err, usr) => {

usr.token = user.token;

done(err, usr);

});

}

* Primero lo que hacemos es llamar nuestra ruta del cliente para la autenticacion, si todo va bien este nos deberia devolver el token generado para este usuario. Luego llamamos **done()**. Si hubo errores le pasamos el mensaje del error.
* Luego si todo salio bien lo que hacemos es hacer el request a nmuestra Db por medio de la ruta que definimos, este nos deberia entregar un error o el usuario completo. Si falla mostramos el error si no le asignamos a usuario el token y retornamos el usuario.
* Ahora tenemos que serializar este usuario ya que no queremos guardar en sesion todo el usuario si no solo una referencia de este. Entonces llamamos a done y luego le pasamos por segundo parametro los campos que queremos serializar. Solo vamos a hacerlo con el username. A este tambien le asignamos el token para guardarlo en la sesion.
* Por ultimo para el deserializar el usuario, lo que hacemos es el proceso inverso. Llamamos la ruta de **getUser()**, le pasamos el user que tenemos en sesion, luego esperamos que este nos devuelva un usuario o un error, si nos devuelve bien el usuario lo que hacemos es a ese usuario que nos entrega asignarle el token y luego llamar a **done()** pasandole un error y el usuario completo.

Ahora tenemos que crear la ruta de login o de autenticacion en express. Veamos como implementamos esto:

app.post('/login', passport.authenticate('local', {

successRedirect: '/',

failureRedirect: '/signin'

}));

function ensureAuth (req, res, next) {

if (req.isAuthenticated()) {

return next();

}

res.status(401).send({error: 'not authenticated'})

}

* Primero para la ruta de login lo que hacemos es pasarle el middleware que nos entrega passport y este nos recibe por parametro la estrategia que se va a usar para validar y la redireccion a la ruta en caso de que sea fallido o que sea exitoso.
* Luego vamos a crear una funcion que nos garantice que el usuario esta autenticado. Este lo usamos para las ruta que tenemos protegidas. Este basicamente lo que hace es usar un metodo de passport que injecta en el request la funcion para saber si esta autenticado o no. Si todo sale bien lo que hace es llamar a next de lo contrario nos arroja un error de autenticacion.

Luego lo que hacemos en la ruta de post/picture es definirle esta funcion como un middleware.

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

Autentucacion con Facebook

Vamos a hacer uso de una estrategia de autenticacion social, en este caso vamos a usar Facebook. Como funcionan estas estrategias sociales?... Estas usar un protocolo llamado **Oauth**, que lo que hace es que nos va a validar la existencia de un usuario con un tercero, esto quiere decir que nosotros le decimos a Facebook que nos entregue informacion de un usuario que existe en su plataforma y Facebook nos devolvera unas Token privados para poder validarnos en nuestra aplicación.

Para poder hacer uso de este recurso tenemos que entrar a la siguiente ruta que nos entrega facebook.

Nos tenemos que autenticar no nuestro perfil de facebook y luego podemos ver las aplicaciones que nosotros hayamos creado.

<https://developers.facebook.com/>

Creamos la aplicación y facebook nos entrega nuestro APP ID y nuestro Secret ID, luego este nos pide que ingresemos el dominio de nuestra aplicación por cuestiones de seguridad, pero como hacemos si nosotros no tenemos aun montada nuestra aplicación? Podemos agregar una extension con .test a la ruta, ejemplo: platzigram.test.

Luego definimos la plataforma en la que vamos a trabajar y posteriormente salvamos los cambios. Listo ahora vamos a usar este AppID y el SecretID para poder hacer uso de la autenticacion con facebook.

Veamos la implementacion de esto. Si no se entiende podemos apoyarnos de la documentacion de passport.JS.

var facebookStrategy = require('passport-facebook').Strategy;

exports.facebookStrategy = new FacebookStrategy({

clientID: config.auth.facebook.clientID,

clientSecret: config.auth.facebook.clientSecret,

callbackURL: config.auth.facebook.callbackURL,

profileFields: ['id', 'displayName', 'email']

}, function (accessToken, refreshToken, profile, done) {

var userProfile = {

username: profile.\_json.id,

name: profile.\_json.name,

email: profile.\_json.email,

facebook: true

}

});

* Primero requerimos el modulo para facebook, luego le pasamos un objeto con los campos que requiere, que son los del APPID y SecretID que nos entrega facebook, por ultimo nos pide la URL del callback que nos va a recibir la informacion que nos va a enviar facebook.
* Por ultimo le decimos que campos queremos recibir de facebook para hacer nuestra autenticacion.
* Luego una funcion que nos recibe un token y un refresh que son los que nos entrega facebook, el primero tiene un tiempo de vida corto, el otro en cambio es mas longevo y nos sirve para volver a pedir un accessToken cuando este haya expirado.
* El siguiente parametro es el perfil del usuario que nos entrega facebook. Y por ultimo el middleware para indicar que todo se completo.

Ahora como estamos manejando un archivo de configuracion para los Tokens que nos entrega facebook, veamos la implementacion de estos metodos y atributos para guardarlos.

auth: {

facebook: {

clientID: process.env.FACEBOOK\_CLIENT\_ID,

clientSecret: process.env.FACEBOOK\_CLIENT\_SECRET,

callbackURL: 'http://platzigram.com/auth/facebook/callback'

}

}

}

if (process.env.NODE\_ENV !== 'production') {

config.client.endpoints = {

pictures: 'http://localhost:5000',

users: 'http://localhost:5001',

auth: 'http://localhost:5002'

}

config.auth.facebook.callbackURL = 'http://platzigram.test:5050/auth/facebook/callback'

}

* Definimos las rutas reales y cuando estemos en un ambiente de desarrollo le definimos una ruta para fines de prueba.

Ahora tenemos que crear la implementacion para validar si este usuario ya existe de lo contrario lo vamos a grabar en nuestra base de datos:

findOrCreate(userProfile, (err, user) => {

return done(null, user);

})

function findOrCreate(usuario, callback) {

client.getUser(user.username, (err, usr) => {

if (err) {

return client.saveUser(user, callback);

}

callback(null, usr);

})

}

});

* Creamos la funcion que nos recibe un callback, luego validamos que si el usuario existe retornemos el usuario, si no existe salvamos el usuario en nuestra base de datos y le pasamos de igual manera un callback.
* Luego llamamos esta implementacion con el usuario que nos devuelve facebook, por ultimo le pasamos el callback con el done que espera la funcion que creamos.

Ahora nos falta implementar un Token con esta estrategia de autenticacion, asi como le entregamos uno cuando nos autenticamos de forma local, lo vamos hacer para este metodo ya que facebook no nos entrega ningun Token.

Creacion de un Token del API a partir de autenticacion por Facebook

Como vimo en la estrategia de autenticacion local, tenemos que generar un Token apartir de la autenticacion, pero en este caso tenemos que instalar la dependencia en nuestro proyecto del Frontend y ademas en nuestro archivo de configuracion vamos a crear un atributo con el secret de los JWT para poder generar un Token correcto.

Entonces instalamos la dependencia de JWT.

**Npm install –save jsonwebtoken**

Ahora veamos la implementacion de esta logica para generar nuestro Token:

findOrCreate(userProfile, (err, user) => {

if (err) return done(err);

jwt.sign({ userId: user.username }, config.secret, {}, (e, token) => {

if (e) return done(e);

user.token = token;

return done(null, user);

})

})

* Primero si nos devuelve un error, retornamos el error. Si no con el metodo que nos entrega JWT firmamos nuestro Token pasandole el campo que queremos firmar o que ira en el Payload del Token y ademas la llave secreta para firmarlo.
* Luego si nos retorna un error, devolvemos un error, de lo contrario asignamos al usuario el Token ya firmado, y ahí es cuando podemos llamar a **done** y pasarle el objeto completo del usuario.
* Por ultimo realizaria la logica de serializar el usuario y en otro caso de deserializarlo.

Ahora tenemos que crear la ruta en express para poder hacer la autenticacion con Facebook, veamos el codigo:

app.get('/auth/facebook', passport.authenticate('facebook', { scope: 'email' }));

* Aca hacemos algo parecido que con la estrategia local, pero marcamos en el metodo de authenticate la estrategia de Facebook, ademas de que le pasamos un objeto con los datos a los que queremos acceder, por ejemplo si queremos hacer uso de mas campos, podemos poner tambien el campo de nombre o de imagen de avatar, pero en este caso solo vamos a usar el email.

Luego tenemos que crear la ruta **callback** donde vamos a recibir los datos que no entrega facebook si todo fue existoso. Veamos la implementacion:

app.get('/auth/facebook/callback', passport.authenticate('facebook', {

successRedirect: '/',

failureRedirect: '/signin'

}))

* Declaramos la ruta con **/callback**, y hacemos de nuevo el llamado al metodo **authenticate()** de passportJS, pero esta vez podemos hacer uso de los atributos de redirect para redireccionar a nuestro usuario si la autenticacion fue exitosa o fallida.

Siguiente a esto tenemos que en nuestros botones del formulario poner el enlace al que tiene que redirigirse y hacer la peticion, veamos el codigo que teniamos en el Frontend:

<a href="/auth/facebook" rel="external" class="btn btn-fb hide-on-small-only">${translate.message('signup.facebook')}</a>

<a href="/auth/facebook" rel="external" class="btn btn-fb hide-on-med-and-up"><i class="fa fa-facebook-official"></i>${translate.message('signup.text')}</a>

* Aca como es una etiqueta **ancla** le ponemos la ruta a la que queremos redirigirnos y ademas tenemos que ponerle un atributo de relacion con valor **External**, esto debido a que nosotros manejamos el patron de SPA (single page aplication) y usamos **pageJS** para hacer la navegacion en nuestra aplicación y si no colocamos este atributo de relacion **pageJS** va a tomar esta ruta del formulario como ruta de navegacion y nosotros queremos es que haga navegacion como tal a una ruta verdadera del navegador, por eso le ponemos este atributo.

Antes de probar esta autenticacion tenemos que hacer un mapeo de nuestras rutas locales, ya que tenemos que hacer que nuestra ip tome valor como si fuera un dominio. Por ejemplo:

**Nuestra ip** 127.0.0.1:5050 -> **platzigram.test**

* Esto para que Facebook nos permita hacer la prueba de su metodo de autenticacion.

Veamos como realizar esto en windows.

Tenemos que buscar la forma de hacerlo en windows ya que no sirve la forma en la que implementa Julian ya que el tiene un sistema operativo tipo UNIX. Bueno una vez que estamos dentro de esto lo que hacemos es crear otra para la ip 127.0.0.1 con el nombre que queremos mapear que seria el **platzigram.test**. Luego si podemos probar la autenticacion pero este nos va a pedir las llaves del APP de facebook. Las tenemos que setear en el archivo que creamos para las variables de entorno.

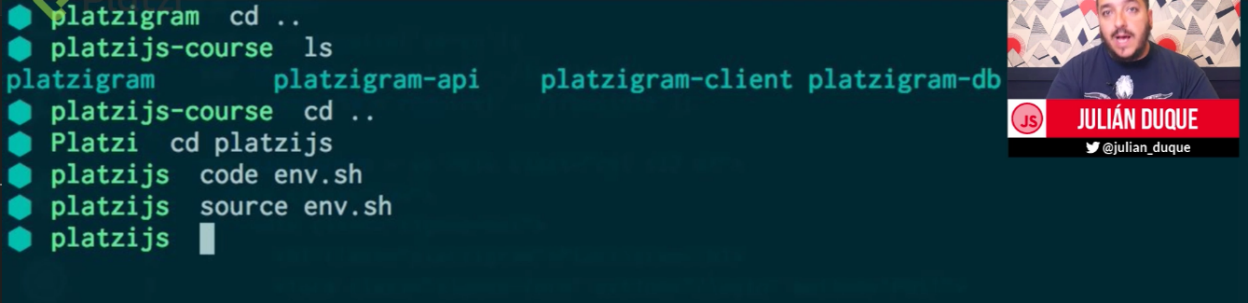
export AWS\_ACCESS\_KEY = "ADWADAFWGG4847GW48"

export AWS\_SECRET\_ACCESS\_KEY = "ADWADAFWGG484AWAWRQQWRQR2R2R27GW48"

export FACEBOOK\_CLIENT\_ID="1945290629058404"

export FACEBOOK\_CLIENT\_SECRET="4744f90fff9b990157c7c703fef905bd"

En este paso no encontre como hacer esto pues lo que hace el profe Julian esta orientado a un sistema operativo tipo UNIX como MAC o Linux. Entonces basicamente lo que hace Julian es setear con el archivo que habiamos creado **env.sh** en variables de entorno del sistema para poder ser usadas en todas las configuraciones que tenemos definidas en nuestro programa.



* En este caso tenemos que hacer este paso manualmente para fines de prueba, pero

Ahora corremos nuestro programa y nos va a arrojar error de que no definimos una estrategia de facebook como autenticacion, esto es por que nosotros nunca definimos el middleware para poder usar la estrategia de facebook. Veamos el codigo:

passport.use(auth.localStrategy);

passport.use(auth.facebookStrategy);

passport.deserializeUser(auth.deserializeUser);

passport.serializeUser(auth.serializeUser);

Ya podemos probar que nos autentique con facebook, correctamente hace la autenticacion y podemos probar haciendo las peticiones a las rutas que tenemos aseguradas, todo deberia pasar con statusCode de 200.

Manejo de sesiones de usuarios

Ahora veamos como podemos hacer para mostrarle a nuestro usuario su nombre si esta autenticado en el header, para esto podemos implementar una ruta en nuestro servidor para recordar que persona esta autenticada en nuestra aplicación, veamos el codigo de esto:

app.get('/whoami', function (req, res) {

if (req.isAuthenticated()) {

return res.json(req.user);

}

res.json({ auth: false })

})

* Hacemos una ruta get y vamos a usar el metodo de **isAuthenticated()** para verificar esto, ademas retornamos un objeto con la informacion del usuario. De lo contrario vamos a retornar un error de autenticacion.

Luego vamos al template del header de nuestro frontend y vamos a hacer los siguientes cambios, vamos a crear unas card de autenticacion:

var yo = require('yo-yo');

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

var empty = require('empty-element');

var authCard = function (ctx) {

var authenticated = yo`

<div class="col s2 m2 push-s10 push-m3">

<a href="#" class="btn btn-large btn-flat dropdown-button" data-activates="drop-user">

<i class="fa fa-user" aria-hidden="true"></i>

${ctx.auth.name}

</a>

<ul id="drop-user" class="dropdown-content">

<li><a href="#">${translate.message('logout')}</a></li>

</ul>

</div>

`

var signin = yo`

<div class="col s2 m2 push-s10 push-m3">

<a href="/signin" class="btn btn-large btn-flat">

${translate.message('signin')}

</a>

</div>

`

if (ctx.auth) {

return authenticated

} else {

return signin

}

}

var renderHeader = function (ctx) {

return yo`<nav class="header">

<div class="nav-wrapper">

<div class="container">

<div class="row">

<div class="col s12 m6 offset-m1">

<a href="/" class="brand-logo platzigram">Platzigram</a>

</div>

${authCard(ctx)}

</div>

</div>

</div>

</nav>`;

}

module.exports = function header(ctx, next) {

var container = document.getElementById('header-container');

empty(container).appendChild(renderHeader(ctx));

next();

}

* Vamos a sacar las secciones donde deberia ir el nombre y ahí decidimos si el usuario esta registrado muestrame el nombre del usuario que esta autenticado pero de lo contrario mostrame un boton para poder logearme.
* Luego vamos a crear una funcion de render para que cuando carguemos el header este llame a la funcion de las card y decida si esta autenticado o no.
* Luego en lugar de pasar el header pasamos la funcion de renderHeader junto al contexto.

Ahora como este componente del header esta siendo utilizado en dos partes, el homepage y la pagina del usuario, tenemos que crear la logica para ajustar y cargar el usuario autenticado, no el que nosotros queremos ver. Veamos como hacer esto:

Vamos a crear un archivo utils para poder crear el request a nuestra ruta whoami de nuestro servidor para poder validar quien esta autenticado. Veamos el codigo…

var axios = require('axios');

async function loadAuth (ctx, next) {

try{

var whoami = await axios.get('/whoami').then(res => res.data)

if (whoami.username) {

ctx.auth = whoami

} else {

ctx.auth = false

}

next()

} catch(err)

{

console.log(err);

}

}

exports.loadAuth = loadAuth

* Aca lo que hacemos es usar axios para hacer peticiones http ya que si usamos fetch lo que pasa es que este no maneja las cookies y nosotros estamos usando cookies en nuestro servidor entonces no nos va a servir la implementacion de este.

Ahora tenemos que implementar este metodo en las dos vistas, la de homepage y la usuario. Veamos como hacerlo:

page('/:user', utils.loadAuth, header, loadUser, function (ctx,next) {

* A estas rutas tanto a las de user como a la de homepage tenemos que definir antes del middleware del header el metodo que exportamos en el utils.js para que este nos cargue el usuario que este logeado. Luego ya podemos probrar y este deberia funcionar correctamente.

Pero en este momento no tenemos la logica para deslogear un usuario, vamos a ver rapidamente como podemos hacer logout usando passport:

Vamos a agregar una ruta de logout en el serevidor.

app.get('/logout', function (req, res) {

req.logout();

res.redirect('/')

})

* Creamos la ruta para el logout, y hacemos uso de un metodo que nos entrega passport para hacer deslogeo y posteriormente podemos hacer el redireccionamiento a la ruta inicial.

Ya funciona todo esto, pero nos falta hacer que nuestro boton de salir este enlazado:

<ul id="drop-user" class="dropdown-content">

<li><a href="/logout" rel="external">${translate.message('logout')}</a></li>

</ul>

* Hacemos que haga la direccion a la externa de nuestro server.

Ahora ya podemos ver como es que vamos a servir todos estos componentes para poder usarlos en conjunto.

Consulta y alta de imágenes

Vamos a implementar los metodos para poder listar las imágenes en relacion al usuario y tambien el metodo para poder guardar la imagen que suba el usuario en el s3 y que se guarde la referencia en la Db. Pero primero vamos a ver como vamos a manejar los avatar en nuestra aplicación, vamos a hacer uso de algo llamado **gravatar**.

<https://es.gravatar.com/>

Este lo que hace es que nos entrega un avatar asociado a un correo electronico. El lugar donde nosotros conocemos el correo de nuestro usuario es en nuestro modulo de API, entonces vamos a instalar esta dependencia en este modulo.

Dentro de nuestra ruta para obtener usuario antes de borrar estos campos hacemos lo siguiente.

hash.set('GET /:username', async function getUser (req, res, params) {

let username = params.username

await db.connect()

let user = await db.getUser(username)

user.avatar = gravatar.url(user.email)

delete user.email

delete user.password

send(res, 200, user)

})

* Gravatar se encarga de codificar el correo que le pasamos por parametro y este nos entrega una url con el avatar del usuario.

Ahora vamos a modificar la ruta para traer las imágenes o para subirlas:

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

upload(req, res, function (err) {

if (err){

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

}

var user = req.user;

var token = req.user.token;

var username = req.user.username;

var src = req.file.location;

client.savePicture({

src: src,

userId: username,

user: {

username: username,

avatar: user.avatar,

name: user.name

}

}, token, function (err, img) {

if (err) return res.status(500).send(err.message);

res.send(`File uploaded: ${req.file.location}`);

})

})

})

* Lo que hacemos es llamar al metodo del api wrapper para subir una foto, le pasamos el token y le pasamos la referencia de la ubicación de la imagen.

Ahora todo deberia funcionar pero lo que pasa es que el formato de fecha que estamos manejando en el frontend no es el correcto y no queremos cambiar toda nuestra api entonces vamos a arreglar esto:

function render(picture) {

return yo`<div class="card ${picture.liked ? 'liked' : ''}">

<div class="card-image">

<img class="activator" src="${picture.src}" ondblclick=${like.bind(null, undefined)} />

<i class="fa fa-heart like-heart ${ picture.likedHeart ? 'liked' : '' }"></i>

</div>

<div class="card-content">

<a href="/${picture.user.username}" class="card-title">

<img src="${picture.user.avatar}" class="avatar" />

<span class="username">${picture.user.name}</span>

</a>

<small class="right time">${translate.date.format(new Data(picture.createdAt).getTime())}</small>

<p>

<a class="left" href="#" onclick=${like.bind(null, true)}><i class="fa fa-heart-o heart-o" aria-hidden="true"></i></a>

<a class="left" href="#" onclick=${like.bind(null, false)}><i class="fa fa-heart heart" aria-hidden="true"></i></a>

<span class="left likes">${translate.message('likes', { likes: picture.likes || 0 })}</span>

</p>

</div>

</div>`

}

* Aca lo que hacemos es ajustar tanto la fecha ya que lo que nos entrega la api es un string y no un timestamp, entonces corregimos eso y tambien corregimos los datos como no los entrega la api.

Ahora ya todo sale funcionando correctamente solo nos faltaria ahora implementar todo para la ruta del perfil de usuario.

Implementacion de perfil de usuario

hash.set('GET /:username', async function getUser (req, res, params) {

let username = params.username

await db.connect()

let user = await db.getUser(username)

user.avatar = gravatar.url(user.email)

let images = await db.getImagesByUser(username)

user.pictures = images

delete user.email

delete user.password

send(res, 200, user)

})

* Cambiamos la logica de nuestra API para que nos cargue las imágenes por usuario para poder renderizarlas en nuestra vista.

Ahora veamos nuestra vista:

var el = yo`<div class="container user-page">

<div class="row">

<div class="col s12 m10 offset-m1 l8 offset-l2 heading">

<div class="row ">

<div class="col s12 m10 offset-m1 l3 offset-l2 center-align">

<img src="${user.avatar}" alt="img" class="responsive-img circle" />

</div>

<div class="col s12 m10 offset-m1 l6 left-align">

<h2 class="hide-on-large-only center-align">${user.name}</h2>

<h2 class="hide-on-med-and-down left-align">${user.name}</h2>

</div>

</div>

<div class="row">

${user.pictures.map(function (picture){

return yo`<div class="col s12 m6 l4">

<a href="/${user.username}/${picture.id}" class="picture-container">

<img src="${picture.src}" class="picture" />

<div class="likes"><i class="fa fa-heart"></i> ${picture.likes || 0}</div>

</a>

<div id="modal${picture.id}" class="modal modal-fixed-footer">

<div class="modal-content center">

<img src="${picture.src}" />

</div>

<div class="modal-footer">

<div class="btn btn-flat likes"><i class="fa fa-heart"></i> ${translate.message('likes', { likes: picture.likes || 0 })}</div>

</div>

</div>

</div>`

})}

</div>

</div>

</div>`;

* Hacemos el mismo procedimiento, arreglamos algunos campos que no estan correctos.

Ya deberia funcionar todo al pelo, ahora solo nos faltaria implementar socket.io para agregar funcionalidad realtime cuando carguemos una imagen en nuestra aplicación y que no hagamos el recargar pagina.

Implementacion de Socket.io junto con RethinkDB

Ahora vamos a implementar Socket.io junto a RethinkDB gracias a que este ultimo tiene un motor de Realtime muy potente. Veamos como es la implementacion de esto. Primero vamos a crear un nuevo modulo para nuestro Realtime Backend y vamos a instalar RethinkDB y Socket.io.

<https://socket.io/docs/>

Vamos a usar en este caso un servidor http de node simple para no tener que instalar de nuevo express.

Tengamos en cuenta de que Socket.io maneja web sockets mediante http, este lo que hace es emitir eventos y recibirlos. RethinkDb tambien implementa estos event emiters y nos facilita el manejar estos eventos que ocurren cuando se actualiza algun dato o tabla de nuestra base de datos.

Veamos el codigo de esta implementacion:

'use strict'

const http = require('http')

const socketio = require('socket.io')

const r = require('rethinkdb')

const config = require('./config')

const server = http.createServer()

const io = socketio(server)

const port = process.env.PORT || 5151

r.connect(config.db, (err, conn) => {

if (err) return console.error(err)

r.table('images').changes().run(conn, (err, cursor) => {

if (err) return console.error(err)

cursor.on('data', data => {

let image = data.new\_val

if (image.publicId != null) {

io.sockets.emit('image', image)

}

})

})

})

server.listen(port, () => console.log(`listening on port ${port}`))

* Creamos el server http con el core que nos entrega node, luego vamos a instanciar socket y le vamos a pasar el servidor que creamos.
* Luego vamos a conectarnos a nuestra base de datos, le pasamos el objeto de configuracion con todos los datos de nuestra base de datos, luego manejamos con callback y recibimos las conexión.
* Ahora vamos a escuchar los cambios en la tabla imágenes. Asi de facil es escuchar en realtime con RethinkDB. Lo que nos entrega RethinkDB es un error o un cursor, este cursor es un iterador, podemos iterar para obtener la informacion que nos llega en cada instancia del iterador.
* Podemos consumir este cursor tambien de forma de event emiters. Le vamos a indicar que queremos escuchar el evento data y este nos entrega un objeto que tiene dos propiedades. Data.new\_val y data.old\_val este *old\_val* es el valor anterior al nuevo dato que ingresa a la base de datos.
* Tenemos que manejar las imágenes por su publicId para no tener que hacer duplicidad de esto ya que nosotros estamos guardando la imagen y luego la estamos actualizando con el publicID, entonces ll que vamos a hacer es que vamos a emitir la imagen solo a las que tengan el publicID.
* Con socket.io usamos el metodo para emitir un evento. Podemos definir un evento custom o un evento predeterminado o creado ya por socket.io en este caso vamos a usar uno custom, el cual le vamos a pasar la imagen que fue creada en la base de datos.
* Por ultimo ponemos a escuchar nuestro servidor con la convencion de nodeJS.

Veamos la configuracion de la base de datos:

'use strict'

const config = {

db: {

host: 'localhost',

port: 28015,

db: 'platzigram'

}

}

module.exports = config

Ahora tenemos que crear nuestro codigo de socket.io para poder recibir y emitir los eventos en Realtime, veamos como implementamos esto en nuestro Frontend.

Escuchar un evento y mostrar los cambios en frontend

Vamos a instalar socketio-client en nuestro frontend (Este socketio-client se puede utilizar tanto en el Backend como en el browser).

**Npm install socket.io-client –save**

Ahora tenemos que crear nuestro cliente en el frontend para escuchar y emitir los cambios que nosotros realicemos en nuestra base de datos y en nuestro frontend.

Tenemos que agregar el siguiente codigo en la pagina de la homepage.

var io = require('socket.io-client')

var picture = requiere('../picture-card')

var socket = io.connect('http://localhost:5151')

socket.on('image', function (image) {

var picturesEl = document.getElementById('pictures-container');

var first = picturesEl.firstChild;

var img = picture(image)

picturesEl.insertBefore(img, first);

});

* Primero requerimos el socket y ademas el formato de card que habiamos creado para poder agregarle estos estilos a nuestra informacion.
* Vamos a conectarnos a nuestro servidor de realtime, esta vez ponemos el localhost ya que estamos probando. Pero despues vamos a ver como podemos configurarlo para apuntar al servidor de produccion.
* Luego vamos a crear un event emiter con el nombre del evento que habiamos definido en nuestro servidor de Realtime y le pasamos la imagen que vamos a recibir o a enviar.
* Ahora vamos a obtener el contenedor HTML de las cards y vamos a obtener el primer elemento de este objeto o arreglo de elemento HTML.
* Luego una vez esto vamos a pasarle a nuestro modulo de picture-cards la imagen para que nos entregue la imagen ya con los estilos y como objeto HTML.
* Para ultimo ya si insertar antes esta nueva imagen gracias al metodo insertBefore() al que le pasamos el elemento a pegar y el elemento primero de esta listade elementos HTML del contenedor.

Esta seria toda nuestra implementacion del modulo de Realtime de nuestro proyecto, ahora vamos a aprender como hacer despliegue de nuestro proyecto, vamos a crear automatizadores que nos van a hacer nuestro proceso de despliegue mas facil y vamos a aprender a utilizar el “wrapper” de todo nuestro proyecto que va a ser NGNX.

Este parte la continuamos en otro archivo word.