Node.js

Parte III

Javier Miguel

- CTO & Freelance Developer
- Email: jamg44@gmail.com
- Twitter: @javiermiguelg

Bases de datos

Bases de datos

Node.js, a través de módulos de terceros, se puede conectar casi con cualquier base de datos del mercado.

Basta con cargar el driver (módulo) adecuado y establecer la conexión.

```
$ npm install mysql
```

```
$ npm install mongoskin
```

Bases de datos - MySQL

Por ejemplo con MySQL:

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
   host     : 'localhost',
   user     : 'usuario',
   password : 'pass',
   database : 'cursonode'
});

connection.connect(); // callback opcional

connection.query('SELECT * from agentes', function(err, rows, fields) {
   if (err) throw err;
   console.log(rows);
});
```

ejemplos/db/mysql

Bases de datos

Para refrescar conceptos de SQL:

https://www.sqlteaching.com/

Bases de datos - SQL ORMs

Un ORM (Object Relational Mapping) se encarga principalmente de:

- Convertir objetos en consultas SQL para que puedan ser persistidos en una base de datos relacional.

- Traducir los resultados de una consulta SQL y generar objetos.

Esto nos resultará util si el diseño de nuestra aplicación es orientado a objetos (OOP).

Bases de datos - SQL ORMs

Un ORM muy usado para bases de datos SQL es sequelize:

http://docs.sequelizejs.com/en/latest/

Sequelize tiene soporte para MySQL, MariaDB, SQLite, PostgreSQL y MSSQL.

Otras buenas alternativas son Knex y Bookshelf.

Ejercicio

Listar agentes en web.

MongoDB

Bases de datos - MongoDB

MongoDB en una base de datos no relacional sin esquemas, esto significa principalmente que:

- No tenemos JOIN, tendremos que hacerlo nosotros
- Cada registro podría tener una estructura distinta
- Mínimo soporte a transacciones

A la hora de decidir que base de datos usar para una aplicación debemos pensar como vamos a organizar los datos para saber si nos conviene usar una base de datos relacional o no relacional.

Bases de datos - MongoDB

Usar una base de datos como MongoDB puede darnos más rendimiento principalmente por alguna de estas razones:

- No tiene que gestionar transacciones
- No tiene que gestionar relaciones
- No es necesario convertir objetos a tablas y tablas a objetos (Object-relation Impedance Mismatch)

Bases de datos - MongoDB

Ejemplo de uso MongoDB:

```
$ npm install mongodb
```

```
var client = require('mongodb').MongoClient;

client.connect('mongodb://localhost:27017/cursonode',
function(err, db) {
  if (err) throw err;
  db.collection('agentes').find({}).toArray(function(err, docs) {
    if (err) throw err;
    console.dir(docs);
    db.close();
  });
});
```

ejemplos/db/mongodb

Bases de datos - MongoDB shell basics

Para acceder a la shell usaremos:

```
~/master/cursonode/mongodb-server/bin/mongo
MongoDB shell version: 3.0.4
connecting to: test
>
```

Bases de datos - MongoDB shell basics

```
show dbs
use <dbname>
show collections
show users
db.agentes.find().pretty()
db.agentes.insert({name: "Brown", age: 37})
db.agentes.remove({_id: ObjectId("55ead8899123383864857 0dd")})
db.agentes.update({_id: ObjectId("55eadb4191233838648570de")}, {$set: {age: 38}})
——————————————————————————————————————————— cuidado con el $set! ——

db.coleccion.drop()
db.agentes.createIndex({name:1, age:-1})
db.agentes.getIndexes()
```

Mas operaciones en la referencia rápida a la shell de MongoDB

Bases de datos - MongoDB queries

```
db.agentes.find({ name : 'Smith'})
db.agentes.find({ _id : ObjectId("55eadb41912338386485 70de")})
db.agentes.find({ age: { $gt: 30}}) // $lt, $gte, $lte, ...
db.agentes.find({ age: { $gt: 30, $lt: 40}});
db.agentes.find({ name: { $in: [ 'Jones', 'Brown']}}) //$nin
db.agentes.find({ name: 'Smith', $or: [
  { age: { $lt: 30}},
  { age: 43 } // 'Smith' and ( age < 30 or age = 43)
] })
```

Bases de datos - MongoDB queries

```
// subdocuments
db.agentes.find({ 'producer.company': 'ACME'})

// arrays
db.agentes.find({ bytes: [ 5, 8, 9 ]}) // array exact
db.agentes.find({ bytes: 5}) // array contain
db.agentes.find({ 'bytes.0': 5}) // array position
```

http://docs.mongodb.org/manual/reference/method/db.collection.find/#db.collection.find

http://docs.mongodb.org/manual/tutorial/query-documents/

Bases de datos - MongoDB queries

Ordenar:

```
db.agentes.find().sort({age: -1})
```

Descartar resultados:

```
db.agentes.find().skip(1).limit(1)
db.agentes.findOne({name: 'Brown'}) // igual a limit(1)
```

Contar:

```
db.agentes.find().count() // db.agentes.count()
```

Bases de datos - MongoDB queries

Full Text Search

Crear índice por los campos de texto involucrados:

```
db.agentes.createIndex({title: 'text', lead: 'text', body: 'text'});
```

Para hacer la búsqueda usar:

```
db.agentes.find({$text:{$search:'smith jones'});
```

Bases de datos - MongoDB queries

Full Text Search

Frase exacta:

```
db.agentes.find({$text:{$search:'smith jones "el elegido"'}});
```

Excluir un término:

```
db.agentes.find({$text:{$search:'smith jones -mister'});
```

Más info:

https://docs.mongodb.com/v3.2/text-search/

https://docs.mongodb.com/v3.2/tutorial/specify-language-for-text-index/

Bases de datos - MongoDB transacción

`findAndModify` es una operación atómica, lo que nos dará un pequeño respiro transaccional.

```
db.agentes.findAndModify({
    query: { name: "Brown"},
    update: { $inc: { age: 1}}
})
```

Lo busca y si lo encuentra lo modifica, no permitiendo que otro lo cambie antes de modificarlo.

Mongoose

Mongoose

Mongoose es una herramienta que nos permite persistir objetos en MongoDB, recuperarlos y mantener esquemas de estos fácilmente.

Este tipo de herramientas suelen denominarse ODM (Object Document Mapper).

Mongoose

Instalación como siempre:

```
npm install mongoose --save
```

diccionario/diccionario-backend

Mongoose

Conectar a la base de datos:

```
var mongoose = require('mongoose');
var conn = mongoose.connection;

conn.on('error', console.error.bind(console, 'mongodb connection
error:'));
conn.once('open', function() {
    console.info('Connected to mongodb.');
});

mongoose.connect('mongodb://localhost/diccionario');?
```

diccionario/diccionario-backend

Mongoose

Crear un modelo:

```
var mongoose = require('mongoose');

var agenteSchema = mongoose.Schema({
    name: String,
    age: Number
});

mongoose.model('Agente', agenteSchema);
```

diccionario/diccionario-backend

Mongoose

Guardar un registro:

```
var agente = new Agente({name: 'Smith', age: 43});

agente.save(function (err, agenteCreado) {
    if (err) throw err;
    console.log('Agente ' + agenteCreado.name + ' creado');
});
```

diccionario/diccionario-backend

Mongoose

Eliminar registros:

```
Agente.remove({ [filters] }, function(err) {
    if (err) return cb(err);
    cb(null);
});
```

diccionario/diccionario-backend

Mongoose

Crear un método estático a un modelo:

```
agenteSchema.statics.deleteAll = function(cb) {
    Agente.remove({}, function(err) {
        if (err) return cb(err);
        cb(null);
    });
};
```

diccionario/diccionario-backend

Mongoose

Crear un método de instancia a un modelo:

```
agenteSchema.methods.findSimilarAges = function (cb) {
  return this.model('Agente').find({ age: this.age }, cb);
}
```

Mongoose

Listando registros:

```
agenteSchema.statics.list = function(cb) {
  var query = Agente.find({});
  query.sort('name');
  query.skip(500);
  query.limit(100);
  query.select('name age');
  return query.exec(function(err, rows) {
      if (err) { return cb(err);}
      return cb(null, rows);
  });
});
```

diccionario/diccionario-backend

Autenticación

HTTP Basic Auth

Autenticación - basic auth

Basic HTTP Auth es un mecanismo sencillo y muy usado en APIS.

```
npm install --save basic-auth
```

Implementaríamos un middleware que comprobara las credenciales.

Autenticación - basic auth

Ventajas:

- Lo soportan todos los navegadores
- Una persona puede realizar la autenticación manualmente de forma sencilla, por ejemplo para explorar el API

Desventajas:

- Hay métodos más seguros

Autenticación - basic auth

```
npm install --save basic-auth

router.use(function(req, res, next) {
  var user = basicAuth(req);
  if (!user || user.name !== 'user' || user.pass !== 'pass') {
    res.set('WWW-Authenticate', 'Basic realm=Authorization
Required');
    return res.send(401);
  }
  next();
};
```

diccionario/diccionario-backend

API Key

Autenticación - passport.js

Ejemplo de API Key con Passport.js

```
$ npm install passport-localapikey
```

```
passport.use(new LocalAPIKeyStrategy(
  function(apikey, done) {
    User.findOne({ apikey: apikey }, function (err, user) {
      if (err) { return done(err); }
      if (!user) { return done(null, false); }
      return done(null, user);
    });
  }
));
router.use(passport.authenticate('localapikey', { session: false,failureRedirect: '/
api/unauthorized' }), function(req, res) {
    res.json({ message: "Authenticated" })
  }
);
```

diccionario/diccionario-backend

JSON Web Token

Autenticación - JWT

JSON Web Token es una forma de tener una sesión, pero guardándola en el cliente.

Cuando se diseña un API para que sea capaz de manejar grandes cantidades de clientes con la menos estructura posible se trata de evitar usar sesiones, porque suponen un coste alto en recursos.

JWT trata de mejorar esto.

Autenticación - JWT

Como funciona:

1. El cliente se autentica
2. El servidor le da un token con info que solo puede abrir el servidor
3. El cliente tiene que enviar el JWT en cada petición
4. El servidor abre el token y comprueba que sigue autenticado

El token viaja en cada petición, por lo cual debemos mantener su contenido con la menor información posible.

diccionario/diccionario-backend

Autenticación - JWT

Crearíamos un controlador con un método de autenticación

```
var jwt = require('jsonwebtoken');

// POST /apiv1/authenticate {user: 'javi', pass: 'asdf'}
router.post('/authenticate', function(req, res) {
  // Buscar al usuario, si existe y la password es correcta
  ...
  // crear un token
  var token = jwt.sign(userFound, config.jwt.secret, {
    expiresInMinutes: config.jwt.expiresInMinutes
  });
}
```

diccionario/diccionario-backend

Autenticación - JWT

Haríamos un middleware que en cada petición abra y compruebe el token, poniendo su contenido en la petición para los siguientes handlers puedan usarlo.

```
var jwt = require('jsonwebtoken');

var token = req.body.token || req.query.token ||
req.headers['x-access-token'];

jwt.verify(token, configJWT.secret, function(err, decoded) {
  req.decoded = decoded;
  next();
}
```

diccionario/diccionario-backend

Buenas prácticas

Node.js

Buenas prácticas - Node.js

Formato de los callbacks con el error siempre como primer parámetro.

```
var resizeImage(imagePath, callBack) {

  // hacer el procesado que queramos
  var resizedImagePath = resizer(imagePath, ...);

  // devolvemos en primer lugar error y luego los resultados
  return callBack(null, resizedImagePath);
}
```

Buenas prácticas - Node.js

Al recibir un callback, comprobar primero si ha habido error para devolverlo y luego hacer el proceso previsto.

```
function getConfig(callBack) {

   leeFichero('./config.json', function(err, fileContent) {
      // primero comprobamos y retornamos posibles errores
      if (err) {
         return callBack(err);
      }

      return callBack(null, JSON.parse(fileContent));
   });

}
```

Buenas prácticas - Node.js

Usar return al invocar callbacks, reduce la posibilidad de errores por olvidar hacer return y llamar varias veces al callback de la función.

```
function saveUpload(res, callBack) {
   leeFichero('./config.json', function(err, fileContent) {
      if (err) {
         callBack(err);
         // aquí deberíamos parar! --> RETURN
         // para no olvidarlo y mejorar la legibilidad usar return callBack(err);
      }
      callBack(null, JSON.parse(fileContent));
      // aquí mejor return callBack(null, JSON...
   });
}
```

Buenas prácticas - Node.js

En los try/catch usar **solo código síncrono dentro del bloque try**.

```
// Parsear JSON
try {
  object = JSON.parse(data);
} catch (exception) {
  return callback(exception);
}

return callback(null, object);
```

Buenas prácticas - Node.js

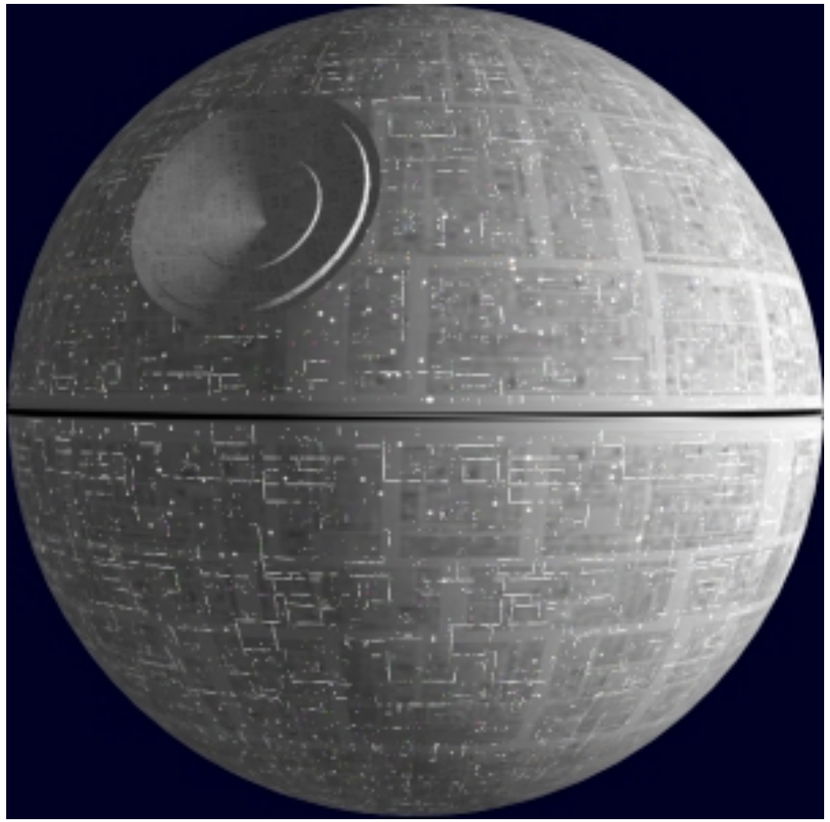
Evitar cuando podamos el uso de this y new.

Normalmente usamos múltiples callbacks para gestionar el flujo de nuestro programa, esto puede hacer que fácilmente nos confundamos de scope.

Buenas prácticas - Node.js

Dividir la funcionalidad en varios módulos especializados, cada uno haciendo solo su trabajo y hacerlo bien.

No construyamos estrellas de la muerte que hacen varias cosas, serán difíciles de mantener!



Buenas prácticas - Node.js

Gestionar lo mejor posible los errores:

- Hay errores operacionales que se pueden reintentar (una petición a un API de terceros que dio timeout) y errores ante los cuales debemos parar la aplicación (no se pudo leer el fichero de configuración)

- Hagamos log de todo!

Buenas prácticas - Node.js

Y algunos consejos más…

- No llamar a una función asíncrona sin pasarle un callback
- Antes de usar propiedades de un objeto pensar si ese objeto puede no existir
- Usar `npm init` al iniciar una aplicación o un módulo nuevo
- Usar patrones reconocibles al procesar de forma asíncrona (p.e. async, promesas)
- Antes de crear un módulo nuevo mirar si ya existe en npm
- Usar ayudantes de estilo como JSCS, JSHint, etc
- Usar NODE_PATH=lib al arrancar nuestra aplicación o apuntar nuestras librerías en el package.json para resolver fácilmente sus rutas
- Documentar las partes de nuestra aplicación con jsdoc o similar
- No poner datos sensibles en el código, usemos ficheros de configuración o variables de entorno

Buenas prácticas

APIs

Buenas prácticas en APIs - desacoplamiento

Nuestro API, aunque lo construyamos inicialmente para un software cliente concreto, debería estar diseñado para que otros tipos de clientes puedan consumirlo.

Por ejemplo GET /homeItems

Estaría mejor como **/offers** al margen de si en una aplicación aparecen en la home o no.

Buenas prácticas en APIs - versionado

Pongamos versión a nuestro API desde el principio.

Cuando tengamos clientes consumiéndola y tengamos que cambiarla tendremos que hacer una nueva versión.

Buenas prácticas en APIs - versionado

Podemos versionar de múltiples formas, las dos más usadas son:

1. Versión en la ruta base /apiV1/users o api/V1/users
2. Versión en la cabecera. El cliente pone la versión que quiere consumir en el header.

La **opción 1** es más visible y por tanto más segura.

Buenas prácticas en APIs - nombres

Hacer un API rest implica exponer recursos que son invocados con GET, POST, PUT, etc.

Los recursos deberían ser nombres, no verbos.

```
GET /users      (mejor que GET /getUserList)
POST /users     (mejor que POST /insertUser)
```

Buenas prácticas en APIs - nombres

GET /users - Devuelve lista de usuarios
GET /users/12 - Devuelve un usuario concreto
POST /users - Crea un usuario
PUT /users/12 - Actualiza usuario #12
PATCH /users/12 - Actualiza parcialmente el usuario #12
DELETE /users/12 - Elimina usuario #12

Solo tenemos que acordarnos del nombre del recurso (users) y la lista de métodos ya nos la sabemos!

Buenas prácticas en APIs - nombres

Singular o plural?

- GET /user
- GET /users

Decisión nuestra, pero usémoslo igual para todos los recursos.

El plural es comúnmente más usado.

Buenas prácticas en APIs - nombres

¿Como hacemos con las relaciones?

GET /users/12/emails - Devuelve lista de emails de usuario
GET /users/12/emails/7 - Devuelve un email concreto
POST /users/12/emails - Crea un email al usuario
PUT /users/12/emails/7 - Actualiza email #7 del usuario
...

Buenas prácticas en APIs - nombres

¿Como hacemos con las acciones distintas a GET, POST, etc?

Si la acción podemos usarla como un campo: PATCH /users/5/active

O podemos usarlo como un sub-recurso. Por ejemplo como hace github PUT /gists/:id/star pone una estrella y DELETE /gists/:id/star la quita.

A veces lo que queremos hacer no encaja con nada o implica múltiples recursos como por ejemplo una búsqueda sobre mensajes, usuarios y teléfonos, para estos casos /search tendrá mucho sentido!

Buenas prácticas en APIs - browser explorability

Facilitar las cosas al consumidor de nuestro API es bueno para que le guste y lo maneja bien.

Tratemos de que la lectura de todos los recursos sea posible desde un navegador.

Por ejemplo, usando el versionado en la ruta, haciendo un índice de los recursos con URLs clickables.

Buenas prácticas en APIs - documentación

Podemos usar herramientas como apidoc, iodocs o swagger para ayudarnos.

Un par de ejemplos de con iodocs:

- https://developers.egnyte.com/io-docs
- http://developer.klout.com/io-docs

Ejemplos de documentación de otras APIs:

- https://swapi.co/documentation

Buenas prácticas en APIs - errores

Un buen API debe devolver errores en un formato estandarizado o previsible.

Los errores de cliente (tipo 400) deberían devolver un objeto JSON con propiedades similares en los distintos casos.

Los errores de servidor (tipo 500) no podrán hacerlo en todos los casos (un balanceador caído es difícil que devuelva JSON), pero al menos los que estén en nuestra mano deberían hacerlo.

Buenas prácticas en APIs - errores

Para los errores de validación donde están involucrados multiples campos pueden devolver algo como:

```
{
  "code" : 1024,
  "message" : "Validation Failed",
  "errors" : [
    {
      "code" : 5432,
      "field" : "first_name",
      "message" : "First name cannot have fancy characters"
    },
    {
       "code" : 5622,
       "field" : "password",
       "message" : "Password cannot be blank"
    }
  ]
}
```

Buenas prácticas en APIs - cambios

Cuando un cliente solicita un cambio en un recurso (con PUT o PATCH) es bastante común que luego tenga que hacer un GET para recuperar los datos guardados y mostrarlos.

Muchos APIs facilitan esto devolviendo siempre una representación del objeto modificado.

Buenas prácticas en APIs - filtros, paginación, etc

Usemos la query-string. Por ejemplo:

Para filtros GET /messages?state=sent&archived=false
Para ordenación GET /messages?sort=-priority,created_at
Para paginación GET /users?skip=30&limit=10&returnTotal=true
Para búsquedas GET /messages?q=frr149&sort=created_at

Buenas prácticas en APIs - alias

Para consultas típicas a nuestro API, quien lo consuma agradecerá que le hagamos alias.

```
GET /messages/recently_sent
```

Buenas prácticas en APIs - reducción de campos

Es muy util permitir al consumidor que elija si quiere menos o más campos de los que devuelve el recurso por defecto.

```
GET /messages?fields=id,subject,send_date,updated_at&state=sent&sort=-updated_at
```

Algunos usan una cabecera como por ejemplo x-extended o similar para obtener todos los campos disponibles. Cuando no se especifica esta cabecera se devolverían los campos de mayor uso.

Buenas prácticas en APIs - HATEOAS o no?

Hypermedia As The Engine Of Application State

Consiste en que nuestro API devuelva links a recursos o acciones que se pueden hacer con los recursos.

```
{
    "content": [ {
        "price": 499.00,
        "description": "Apple tablet device",
        "name": "iPad",
        "links": [ {
            "rel": "self",
            "href": "http://localhost:8080/product/1"
        } ],
        "attributes": {
            "connector": "socket"
        }
    }
}
```

Buenas prácticas en APIs - timestamps

Permitir solicitar al API los elementos que hayan cambiado desde la última vez que los descargamos es muy bueno para que las apps usen el menor ancho de banda posible y la experiencia sea mejor.

Esto lo conseguimos devolviendo el timestamp actual en cada lista, e implementando un filtro opcional al que pasarle dicho timestamp para que el API nos de solo los elementos que hayan cambiado desde el momento del timestamp.

Los elementos que nos dará serán o nuevos (tendrán un ID que nosotros no tenemos), actualizados (los localizamos por el ID) o borrados (localizados por ID y llevarán una marca de 'DELETED' para que los borremos).

Buenas prácticas en APIs - method override

Hay clientes que solo pueden usar GET y POST por ejemplo.

Por tanto no podrán llamar a nuestros PUT, PATCH o DELETE.

Para solucionar esto habilitamos una cabecera llamada '*X-HTTP-Method-Override*' (por convención) con un string que contendrá uno de los métodos PUT, PATCH or DELETE.

Usemos esto solo para POST! (los métodos GET nunca deben hacer modificaciones!)

Buenas prácticas en APIs - rate limiting

Para evitar abusos del API se suele implementar algún sistema que controle e informe al consumidor sobre los límites de llamada al API.

Hay muchos módulos de terceros que nos darán estas habilidades, como por ejemplo:

- https://github.com/jhurliman/node-rate-limiter
- https://github.com/ded/express-limiter

Buenas prácticas en APIs - CORS

Aunque los primeros clientes de nuestro API sean aplicaciones móviles nativas, es bastante posible que más adelante algún cliente web tenga que consumirlo. Para facilitarlo, habilitemos CORS en nuestro servicio.

http://enable-cors.org/server_expressjs.html

```
app.use(function(req, res, next) {
  res.set("Access-Control-Allow-Origin", "*");
  res.set("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
  res.set("Access-Control-Allow-Headers", "Origin, X-Requested-With,
Content-Type, Accept");
  next();
});
```

Buenas prácticas en APIs - HTTPS

HTTPS everywhere - all the time!

Nuestra API va a ser consultada desde wifis de aeropuertos, bares, bibliotecas, etc.

A veces se implementa que una petición HTTP redirige a la misma con HTTPS. No hagamos esto, en vez de eso devolvamos un error, hagamos saber claramente **que nuestra API no usa métodos inseguros**.