Creación de una API

Creación de una API con node.js

Primeros pasos

- npm configurado y git configurado
- Crear repositorio en GitHub
- Clonar repositorio
- Ejecutar npm init
- Crear la estructura de la aplicación, por el momento una carpeta app donde iremos guardando el código propio de la aplicación

Configuración de eslint

- Si tienes los linters para Sublime y js configurados, comprueba que la consola arroja un error.
- Hay que configurar eslint para formatear nuestro código.

```
npm i -D eslint
```

- Como lo instalo en local, para ejecutarlo, necesito darle el path:

```
./node_modules/.bin/eslint --init
? How would you like to configure ESLint? Answer questions about your style
? Are you using ECMAScript 6 features? No
? Where will your code run? Node
? Do you use JSX? No
? What style of indentation do you use? Spaces
```

```
? What quotes do you use for strings? Single
? What line endings do you use? Unix
? Do you require semicolons? No
? What format do you want your config file to be in? JSON
Successfully created .eslintrc.json file in /home/juanda/api_node_express/ejercicio3-nodemon-eslint
```

- Comprueba que el linter de JavaScript te funcione bien

express

- Utilizaremos express para realizar la API
- Instalar express mediante uno de los comandos siguientes:

```
npm install --save express
npm i -S express
```

- Creamos el fichero app/server.js donde pondremos el código necesario para testear una API muy básica para probar Express.

https://www.guru99.com/images/NodeJS/010716_0613_NodejsExpre1.png

- Utiliza el plugin ExpressComplete (aget, aput...) de Sublime para autocompletado

```
var express = require('express') //llamamos a Express
var app = express()

var port = process.env.PORT || 8080  // establecemos nuestro puerto

app.get('/', function(req, res) {
  res.json({ mensaje: '¡Hola Mundo!' })
})

app.get('/cervezas', function(req, res) {
  res.json({ mensaje: '¡A beber cerveza!' })
})

app.post('/', function(req, res) {
  res.json({ mensaje: 'Método post' })
})
```

```
app.del('/', function(req, res) {
  res.json({ mensaje: 'Método delete' })
})

// iniciamos nuestro servidor
app.listen(port)
console.log('API escuchando en el puerto ' + port)
```

- Si el linter de js te da un error por usar *console.log*, puedes deshabilitar esa regla en el fichero de configuración de eslint

```
no-console: 0
```

Iniciar y testear a mano nuestra API

- Iniciamos nuestro API Server mediante el comando

```
node app/server.js
```

- Probamos que la API funcione mediante http://localhost:8080 o utilizando la extensión de Google Chrome **Postman**
- Creamos una entrada en nuestro fichero package.json, de modo que podamos arrancar nuestra API mediante `npm start`

```
"start": "node app/server.js"
```

- Hagamos un commit del repositorio, pero sin tener en cuenta la carpeta node_modules.
- Comprueba con el segundo *git status* que git no lo tiene en cuenta antes de continuar con el commit:

```
git status
echo "node_modules">.gitignore
git status
git add -A *
git commit -m "Primera versión API"
git push
```

- Debemos hacer nuevas instantáneas en pasos posteriores, pero ya no las documentaré por brevedad.

nodemon

- Es un wrapper de node, para reiniciar nuestro API Server cada vez que detecte modificaciones.

```
npm i -D nodemon
```

- Cada vez que ejecutemos **npm start** ejecutaremos nodemon en vez de node. Habrá que cambiar el script en el fichero *package.json*:

```
"start": "nodemon app/server.js"
```

Uso de enrutadores

- Imagina que nuestra API es compleja:
  - Tiene varios recursos (GET, POST... por cada uno de ellos)
  - Versionado de la API
- Utilizaremos enrutadores
  - Asociamos enrutadores a la app en vez de rutas directamente
  - Cada enrutador se puede asociar por ejemplo a un recurso
  - Se pueden anidar enrutadores
- El código para un enrutador sería así:

```
// para establecer las distintas rutas, necesitamos instanciar el express router
var router = express.Router()

//establecemos nuestra primera ruta, mediante get.
router.get('/', function(req, res) {
  res.json({ mensaje: '¡Bienvenido a nuestra API!' })
})

// nuestra ruta irá en <http://localhost:8080/api>
// es bueno que haya un prefijo, sobre todo por el tema de versiones de la API
app.use('/api', router)
```

Recibir parámetros

- Cuando el router recibe una petición, podemos observar que ejecuta una función de callback:

```
function (req, res){}
```

  - El parámetro **req** representa la petición (request)
  - El parámetro **res** representa la respuesta (response)
  - En el caso anterior hemos codificado la respuesta en json:

```
res.json({ mensaje: '¡Bienvenido a nuestra API!' })
```

- A menudo la petición se hará enviando algún parámetro adicional. Hay varias posibilidades:
  - Mediante la url: se recogerán mediante

```
req.param.nombreVariable
```

  - Mediante post en http hay dos posiblidades:
    - application/x-www-form-urlencoded
    - multipart/form-data
- En peticiones post, escogeremos x-www-form-urlencoded si no se envía una gran cantidad de datos (normalmente ficheros)

Parámetros por url

- Vamos a mandar un parámetro *nombre* a nuestra api, de modo que nos de un saludo personalizado.

```
router.get('/:nombre', function(req, res) {
  res.json({ mensaje: '¡Hola' + req.params.nombre })
})
```

Parámetros por post

- Parámetros mediante **POST y application/x-www-form-urlencoded**:
  - Necesitaremos body-parser: extrae los datos del body y los convierte en json
- Parámetros mediante **POST y multipart/form-data**
  - Usaremos https://www.npmjs.com/package/busboy o Multer

Ejemplo con body-parser

- Hay que instalar body-parser

```
npm i -S body-parser
```

- body-parser actúa como **middleware**
- El código adicional será similar al siguiente:

```
var bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

router.post('/',function(req,res) {
  res.json({mensaje: req.body.nombre})
})
```

Rutas de nuestra API

- Las rutas que utilizaremos son las siguientes:

Ruta	Verbo http	Descripción
/api/cervezas	GET	Obtenemos todas las cervezas
/api/cervezas/search?q=keyword	GET	Obtenemos cervezas por keyword
/api/cervezas/:id	GET	Obtenemos los datos de una cerveza
/api/cervezas	POST	Damos de alta una cerveza
/api/cervezas/:id	PUT	Actualizamos los datos de una cerveza
/api/cervezas/:id	DELETE	Borramos los datos de una cerveza

- Como la configuración se complica, lo suyo es dividirla:
  - Creamos una carpeta app/routes donde irán las rutas
    - Creamos un fichero *app/routes/index.js* donde irá el enrutador del versionado de la API
    - Creamos un fichero en *app/routes* por cada resouce (en este caso solo uno, *cervezas.js*)
- El fichero *app.js* queda así:

```
var express = require('express') //llamamos a Express
var app = express()
var bodyParser = require('body-parser')

var port = process.env.PORT || 8080  // establecemos nuestro puerto

app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

// nuestra ruta irá en <http://localhost:8080/api>
// es bueno que haya un prefijo, sobre todo por el tema de versiones de la API
var router = require('./routes')
app.use('/api', router)

//arrancamos el servidor
app.listen(port)
console.log('API escuchando en el puerto ' + port)
```

- El fichero *app/routes/index.js*

```
var router = require('express').Router()
var cervezas = require('./cervezas')

router.use('/cervezas', cervezas)

router.get('/', function (req, res) {
  res.status(200).json({ message: 'Estás conectado a nuestra API' })
})

module.exports = router
```

- Y el fichero *app/routes/cervezas.js*:

```
var router = require('express').Router()
router.get('/search', function(req, res) {
  res.json({ message: 'Vas a buscar una cerveza' })
})
router.get('/', function(req, res) {
  res.json({ message: 'Estás conectado a la API. Recurso: cervezas' })
```

```
})
router.get('/:id', function(req, res) {
  res.json({ message: 'Vas a obtener la cerveza con id ' + req.params.id })
})
router.post('/', function(req, res) {
  res.json({ message: 'Vas a añadir una cerveza' })
})
router.put('/:id', function(req, res) {
  res.json({ message: 'Vas a actualizar la cerveza con id ' + req.params.id })
})
router.delete('/:id', function(req, res) {
  res.json({ message: 'Vas a borrar la cerveza con id ' + req.params.id})
})
module.exports = router
```

Acceso a base de datos

- Para la persistencia de nuestros datos utilizaremos una base de datos
- Optamos por una base de datos MongoDB:
  - Es lo más habitual en arquitecturas MEAN
  - Así operamos con objetos json tanto en node como en bbdd (bson)
  - Nos permite más libertad, al utilizar colecciones en vez de tablas

Instalación de MongoDB

- Instalamos y leventamos el servicio de MongoDB:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
echo "deb <http://repo.mongodb.org/apt/ubuntu> trusty/mongodb-org/3.2 multiverse" |
sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list
sudo apt-get update
sudo apt-get install -y mongodb-org
```

- El servicio se levanta como otros servicios de Linux:

```
sudo service mongod start
```

- Y para entrar a su consola, mediante **mongo**, o mediante algún gui como por ejemplo Robomongo
- La consola de Mongo también es un intérprete de JavaScript :-)

Inserción de datos

- Utilizaremos el fichero *cervezas.json*, lo podemos obtener mediante:

```
wget
<https://raw.githubusercontent.com/juanda99/proyecto_web_basica/master/cervezas.json>
```

- Importar nuestro cervezas.json a una base de datos

```
mongoimport --db web --collection cervezas --drop --file cervezas.json --jsonArray
```

- Para poder hacer una búsqueda por varios campos de texto, tendremos que hacer un índice:

```
$ mongo # para entrar en la consola de mongo
use web; #seleccionamos la bbdd
db.cervezas.createIndex(
 {
   "Nombre": "text",
   "Descripción": "text"
 },
 {
   name: "CervezasIndex",
   default_language: "spanish"
 }
)
```

- Comprobamos que el índice esté bien creado

```
db.cervezas.getIndexes()
```

- Si hiciera falta, lo podemos recrear:

```
db.cervezas.dropIndex("CervezasIndex")
```

Instalación de Mongoose

- Instalaremos [mongoose] como ODM (Object Document Mapper) en vez de trabajar con el driver nativo de MongoDB (se utiliza por debajo).

```
npm i -S mongoose
```

Uso de Mongoose

- Creamos el fichero *app/db.js* donde configuraremos nuestra conexión a base de datos mediante mongoose:

```
//incluimos Mongoose y abrimos una conexión
var mongoose = require('mongoose')
var MONGO_URL = process.env.MONGO_URL || 'mongodb://localhost/web'
mongoose.connect(MONGO_URL)

mongoose.connection.on('connected', function () {
  console.log('Conectado a la base de datos: ' + MONGO_URL)
})

mongoose.connection.on('error',function (err) {
  console.log('Error al conextar a la base de datos: ' + err)
})

mongoose.connection.on('disconnected', function () {
  console.log('Desconectado de la base de datos')
})

process.on('SIGINT', function() {
  mongoose.connection.close(function () {
    console.log('Desconectado de la base de datos al terminar la app')
    process.exit(0)
  })
})
```

- En nuestro fichero *app/server.js* incluimos el fichero de configuración de bbdd:

```
var express = require('express') //llamamos a Express
var bodyParser = require('body-parser')
/*toda la configuración de bbdd la hacemos en un fichero a parte*/
```

```
require('./db')
....
```

Modelos

- Definimos un esquema para nuestros objetos (cervezas) y creamos nuestro modelo a partir del esquema (fichero *app/models/Cervezas.js*):

```
var mongoose = require('mongoose')
var Schema = mongoose.Schema

var cervezaSchema = new Schema({
  Nombre: String,
  Descripción: String,
  Graduacion: String,
  Envase: String,
  Precio: String
})

var Cerveza = mongoose.model('Cerveza', cervezaSchema)

module.exports = Cerveza
```

- Ahora podríamos crear documentos y guardarlos en la base de datos:

```
var miCerveza = new Cerveza({ name: 'Ambar' });
miCerveza.save(function (err, miCerveza) {
  if (err) return console.error(err);
  console.log('Guardada en bbdd' + miCerveza.name);
})
```

Uso de controladores

- Desde nuestro fichero de rutas (*app/routes/cervezas.js*), llamaremos a un controlador que será el encargado de añadir, borrar o modificar cervezas en base al modelo anterior.
- Nuestro código queda así perfectamente separado y cada fichero de rutas se encargará solo de gestionar los endpoints de nuestra API para el recurso en cuestión.

- Creamos un directorio específico para los controladores, donde colocaremos el específico para las cervezas.
- Nuestro código quedas así (fichero *app/controllers/cervezasController.js*):

```
var Cervezas = require('../models/Cervezas')
module.exports = {
// <https://docs.mongodb.com/v3.0/reference/operator/query/text/>
search: function (req, res) {
  var q = req.query.q
  Cervezas.find({ $text: { $search: q } }, function(err, cervezas) {
    if(err) {
      return res.status(500).json({
        message: 'Error en la búsqueda'
      })
    }
    return res.json(cervezas)
  })
},
list: function(req, res) {
  Cervezas.find(function(err, cervezas){
    if(err) {
      return res.status(500).json({
        message: 'Error obteniendo la cerveza'
      })
    }
    return res.json(cervezas)
  })
},
show: function(req, res) {
  var id = req.params.id
  Cervezas.findOne({_id: id}, function(err, cerveza){
    if(err) {
      return res.status(500).json({
        message: 'Se ha producido un error al obtener la cerveza'
      })
    }
    if(!cerveza) {
      return res.status(404).json( {
        message: 'No tenemos esta cerveza'
      })
```

```
    }
    return res.json(cerveza)
  })
},
create: function(req, res) {
  var cerveza = new Cervezas (req.body)
  cerveza.save(function(err, cerveza){
    if(err) {
      return res.status(500).json( {
        message: 'Error al guardar la cerveza',
        error: err
      })
    }
    return res.status(201).json({
      message: 'saved',
      _id: cerveza._id
    })
  })
},
update: function(req, res) {
  var id = req.params.id
  Cervezas.findOne({_id: id}, function(err, cerveza){
    if(err) {
      return res.status(500).json({
        message: 'Se ha producido un error al guardar la cerveza',
        error: err
      })
    }
    if(!cerveza) {
      return res.status(404).json({
        message: 'No hemos encontrado la cerveza'
      })
    }
    cerveza.Nombre = req.body.nombre
    cerveza.Descripción =  req.body.descripcion
    cerveza.Graduacion = req.body.graduacion
    cerveza.Envase = req.body.envase
    cerveza.Precio = req.body.precio
    cerveza.save(function(err, cerveza){
```

```
      if(err) {
        return res.status(500).json({
          message: 'Error al guardar la cerveza'
        })
      }
      if(!cerveza) {
        return res.status(404).json({
          message: 'No hemos encontrado la cerveza'
        })
      }
      return res.json(cerveza)
    })
  })
},
remove: function(req, res) {
  var id = req.params.id
  Cervezas.findByIdAndRemove(id, function(err, cerveza){
    if(err) {
      return res.json(500, {
        message: 'No hemos encontrado la cerveza'
      })
    }
    return res.json(cerveza)
  })
}
}
```

- El router que gestiona el recurso se encarga de llamarlo (fichero *app/routes/cervezas.js*):

```
var router = require('express').Router()
var cervezasController = require ('../controllers/cervezasController')

router.get('/search', function(req, res) {
  cervezasController.search(req, res)
})
router.get('/', function(req, res) {
  cervezasController.list(req, res)
})
router.get('/:id', function(req, res) {
  cervezasController.show(req, res)
```

```
})
router.post('/', function(req, res) {
  cervezasController.create(req, res)
})
router.put('/:id', function(req, res) {
  cervezasController.update(req, res)
})
router.delete('/:id', function(req, res) {
  cervezasController.remove(req, res)
})
module.exports = router
```

Test desde el navegador o mediante Postman

- Comprobamos que se genera el listado de cervezas
- Comprobamos que se busca por keyword:...

```
<http://localhost:8080/api/cervezas/search?q=regaliz>
```

Test de la API

- Utilizaremos Mocha como test framework y supertest para hacer las peticiones http.

```
npm i -D mocha supertest
```

- Creamos nuestro fichero *tests/api.test.js* con la prueba para crear una cerveza:

```
'use strict'
/* global describe it */
var request = require('supertest')

/*obtenemos nuestra api rest que vamos a testear*/
var app = require('../app/server')

describe('Crear una nueva cerveza', function() {
```

```
  it('Crea la cerveza retornando 201', function(done) {
    request(app)
      .post('/api/cervezas/')
      .set('Accept', 'application/json')
      .expect('Content-Type', /json/)
      .send({
        'Nombre': 'DAMN',
        'Descripción': 'Mi cerveza preferida',
        'Graduación': '10º',
        'Envase': 'Bidón',
        'Precio': '1 eurito'
      })
      .expect(201, done)
  })
})
```

- Utiliza los paquetes **Mocha Snippets** y **Chai Completions** de Sublime Text para completar el código
- Si echamos un vistazo al código anterior:
- **describe** nos sirve para describir los test suites (se pueden anidar varios)
  - **it** nos sirve para describir cada caso de test.
  - **requests(app).post** realizará una petición post a nuestra api
- ¿Cómo ejecutamos el test?
  - Mediante línea de comandos (hace falta la ruta completa ya que no hemos instalado el paquete de forma global):

```
node_modules/.bin/mocha tests
```

  - Introduciendolo en el fichero package.json (no hace falta la ruta al estar dentro del package.json):

```
"test": "mocha tests"
```

- Para que el caso anterior funcione, tenemos que modificar nuestro fichero *app/server* de modo que se pueda utilizar *app* desde otro fichero js:

```
/*lo añado al final de app/server.js:*/
module.exports = app
```

- Por último podríamos utilizar un paquete como **istanbul** que nos analice el código y ver si nuestras pruebas recorren todas las instrucciones, funciones o ramas del código:

```
npm i -D istanbul
./node_modules/.bin/istanbul cover -x "**/tests/**"  ./node_modules/.bin/_mocha
tests/api.test.js
```

- Estos datos son facilmente exportables a algún servicio que nos de una estadística de la cobertura de nuestros tests o que haga un seguimiento de los mismos entre las distintas versiones de nuestro código.
- Por último también se podría integrar con un sistema de integración continua tipo Travis.

Uso de middlewares

- Son funciones que tienen acceso al objeto de solicitud (req), al objeto de respuesta (res) y a la siguiente función de middleware en el ciclo de solicitud/respuestas de la aplicación.
- La siguiente función de middleware se denota normalmente con una variable denominada next.
- Podemos crear un middleware que guarde traza de las fechas de acceso:

```
var app = express();

app.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});
```

- Normalmente utilizaremos middlewares que ya están hechos, por ejemplo Morgan para logs y cors para Cors.
- Los instalamos:

```
npm i -S cors morgan
```

- Los insertamos en nuestra API (el orden puede ser importante):

```
var express = require('express') //llamamos a Express
var app = express()
var cors = require('cors')
var bodyParser = require('body-parser')
var morgan = require('morgan')

var port = process.env.PORT || 8080  // establecemos nuestro puerto

/*toda la configuración de bbdd la hacemos en un fichero a parte*/
require('./db')
```

```
app.use(morgan('combined'))
app.use(cors())
app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

// para establecer las distintas rutas, necesitamos instanciar el express router
var router = require('./routes')
app.use('/api', router)

// iniciamos nuestro servidor
app.listen(port)
console.log('API escuchando en el puerto ' + port)

/*lo añado al final de app/server.js:*/
module.exports = app
```

Revision #1
Created 19 April 2023 18:07:08 by molombo
Updated 19 April 2023 18:12:03 by molombo