Aplicaciones RESTful:
Implementación en Node / Express

Teoría

Introducción

NodeJS cuenta con módulos nativos para manejar el envío y recepción de peticiones de tipo http/s, sin embargo, usaremos para nuestra aplicación un módulo externo, muy popular y fácil de usar, que nos facilitará la tarea de crear los distintos puntos de entrada de nuestro servidor, así como también personalizar la forma en que se manejará cada petición en forma más simple y rápida.

Express.js

Express es un framework web minimalista, con posibilidad de ser utilizado tanto para aplicaciones / páginas web como para aplicaciones de servicios. Como todo módulo, lo primero que debemos realizar es agregarlo como dependencia en nuestro proyecto:

Instalación desde la consola

```
$ npm install --save express
```

Express como framework soporte para servidores REST

Express nos permite definir, para cada tipo de petición HTTP que llegue a una determinada URL, qué acciones debe tomar, mediante la definición de un callback para cada caso que consideremos necesario incluir en nuestra API.

Uso del módulo

Para poder usar el módulo, lo primero que debemos hacer es importarlo al comienzo de nuestro archivo. El objeto obtenido luego del *import* es una función que al ejecutarla nos devuelve el servidor que configuraremos posteriormente con los detalles de nuestra aplicación.

Ejemplo de inicialización

```
import express from 'express'
const app = express()
```

Conexión del servidor

Otra cosa que debemos hacer es indicar en qué puerto de nuestra computadora queremos que nuestra aplicación comience a escuchar peticiones. Este puerto será de uso exclusivo de nuestro servidor, y no podrá ser compartido con otras aplicaciones.

Ejemplo de conexión

```
const puerto = 8080

const server = app.listen(puerto, () => {
    console.log(`servidor inicializado en puerto ${server.address().port}`)
})
```

Si el puerto elegido es el cero (0), *express* elegirá un puerto al azar entre los disponibles del sistema operativo en ese momento.

Manejo de errores

Es posible que el servidor no se conecte correctamente por algún motivo. Es una buena práctica definir cuál debe ser el comportamiento del programa en caso de que esto sucediera. Para ello, el servidor (al igual que otros objetos dentro del ecosistema de JS, aunque no vamos a entrar más en detalles) posee la capacidad de recibir mensajes o "eventos" del sistema. Es así que existen distintos tipos de eventos, y vamos a concentrarnos puntualmente en dos de ellos:

"listening": este evento se dispara cuando el servidor se conecta exitosamente al puerto que corresponde, y pasa a estar listo para escuchar peticiones.

"error": este evento se dispara cuando el servidor no logra conectarse por algún motivo.

La manera de reaccionar ante estos eventos es definir un *manejador de eventos* para cada uno en particular. Es decir, vamos a decirle al servidor qué debe hacer cuando recibe un evento de tipo *listening* y qué debe hacer cuando recibe un evento de tipo *error*. Esto se hace a través del método **on**, y la sintaxis es la siguiente:

```
const server = app.listen(puerto, () => {
    console.log('conectado!')
})
server.on('error', error => { console.log(error.message) })
```

Es callback del app.listen maneja automáticamente el evento *'listening*', pero si quisiéramos hacerlo más explícito, el siguiente código es equivalente:

```
const server = app.listen(puerto)
server.on('listening', () => { console.log('conectado!') })
server.on('error', error => { console.log(error.message) })
```

Configuración extra

Para que nuestro servidor express pueda interpretar mensajes de tipo JSON al recibirlos, en forma automática, debemos indicarlo en forma explícita, agregando la siguiente línea luego de crearlo. Sin esta línea, el servidor no sabrá cómo interpretar los objetos recibidos!

```
app.use(express.json())
```

Manejo de peticiones

Para definir cómo se debe manejar cada tipo de petición usaremos los métodos nombrados de acuerdo al tipo de petición que manejan: get(), post(), delete(), y put().
Todos ellos reciben como primer argumento la ruta que van a estar escuchando, y solo manejarán ***peticiones que coincidan en ruta y en tipo***. Luego, el segundo argumento será el callback con que se manejará la petición. Está tendrá dos parámetros, el primero con la petición (request) en sí, y el segundo con la respuesta (response) que espera devolver.

Cada tipo de petición puede tener diferentes características. Por ejemplo, algunas peticiones no requieren el envío de ningún dato extra en particular para obtener el recurso buscado. Este es el caso de la petición GET. Como respuesta a la petición, devolveré el resultado deseado en forma de objeto.

Ejemplo de petición GET

```
app.get('/api/mensajes', (req, res) => {
    console.log('request recibido')

    // acá debería obtener todos los recursos de tipo ‘mensaje’

    res.json({ msg: 'Hola mundo!'})
})
```

Por otra parte, en ocasiones nos quieren dar más detalles sobre la búsqueda que se quiere realizar, por ejemplo, enviando parámetros adicionales para refinar la misma. En estos casos, estos parámetros se agregan al final de la URL, indicando el comienzo de los mismos usando

un signo de interrogación ‘?’ y enumerando pares ‘clave=valor’, separando entre sí cada par por un ampersand ‘&’ en el caso de que hubiere más de uno. A la hora de recibirlos, los mismos se encontrarán en el objeto ‘query’ dentro del objeto petición (req).

Ejemplo de petición GET con parámetros de búsqueda

```
app.get('/api/mensajes', (req, res) => {
    console.log('GET request recibido')

    if (Object.entries(req.query).length > 0) {
        res.json({
            result: 'get with query params: ok',
            query: req.query
        })
    } else {
        res.json({
            result: 'get all: ok'
        })
    }
})
```

En este ejemplo, utilizo un método de la clase Object que me devuelve un array con todas las propiedades del objeto en cuestión (req.query) para verificar si éste está vacío, lo cual indicaría que no se hubo envíado ningún parámetro adicional en la URL.

Finalmente, en el caso de que se quiera acceder a un recurso en particular, ya conocido, es necesario enviar algún identificador unívoco en la URL que lo identifique. Para enviar este tipo de parámetros, el mismo se escribirá luego del nombre del recurso (en la URL), separado por una barra. Por ejemplo:

```
http://miservidor.com/api/mensajes/1
```

En este ejemplo, estoy queriendo acceder al mensaje nro 1 de mis recursos.

Para acceder a este campo identificador, desde el lado del servidor, express utiliza una sintaxis que permite indicar anteponiendo ‘dos puntos’ antes del nombre del campo identificador, al especificar la ruta escuchada. Luego, para acceder al valor del mismo, se hará a través del campo ‘params’ del objeto petición recibido en el callback.

Ejemplo de petición GET con identificador

```
app.get('/api/mensajes/:id', (req, res) => {
    console.log('GET request recibido')

    // acá debería hallar y devolver el recurso con id == req.params.id

    res.json(elRecursoBuscado)
})
```

Las demás peticiones se manejan de manera similar. Por ejemplo, si quisiéramos eliminar un recurso, debemos identificar unívocamente sobre cuál de todos los disponibles se desea realizar la operación.

Ejemplo de petición DELETE

```
app.delete('/api/mensajes/:id', (req, res) => {
    console.log('DELETE request recibido')

    // acá debería eliminar el recurso con id == req.params.id

    res.json({
        result: 'ok',
        id: req.params.id
    })
})
```

Algunas peticiones, en cambio, requieren el envío de algún dato desde el cliente hacia el servidor. Por ejemplo, al crear un nuevo registro. Este es el caso de la petición POST.
Para acceder al cuerpo del mensaje, incluído en la petición, lo haremos a través del campo ‘body’ del objeto petición recibido en el callback. En este caso, estoy devolviendo como respuesta, el mismo registro que se envió en la petición.

Ejemplo de petición POST

```
app.post('/api/mensajes', (req, res) => {
    console.log('POST request recibido')

    // acá debería crear y guardar un nuevo recurso
    // const mensaje = req.body
```

```
    res.json({
        result: 'ok',
        body: req.body
    })
})
```

Como es de esperarse, también es posible mezclar varios mecanismos de pasaje de datos/parámetros, como es el caso de las peticiones de tipo PUT, en las que se desea actualizar un registro con otro nuevo provisto por el cliente. Para ello, se debe proveer el identificador del registro a reemplazar, y el dato con el que se lo quiere sobreescribir.

Ejemplo de petición PUT

```
app.put('/api/mensajes-json/:id', (req, res) => {
    console.log('PUT request recibido')

    // acá debo hallar al recurso con id == req.params.id
    // y luego reemplazarlo con el registro recibido en req.body

    res.json({
        result: 'ok',
        id: req.params.id,
        nuevo: req.body
    })
})
```

Apéndice A - Creación de servidores usando promesas

Como hemos visto, la manera de manejar el asincronismo en express es mediante el uso de callbacks. Es decir que si quisiéramos crear un servidor y hacer algo después de que se haya conectado, debemos escribirlo dentro del callback correspondiente. Si bien express no ofrece una opción de fábrica que utilice promesas, podemos construir nuestra propia versión, utilizando el siguiente código.

```
const promesaDeServidorConectado = new Promise((resolve, reject) => {
  const server = app.listen(port)
  server.on('listening', () => { resolve(server) })
  server.on('error', error => { reject(error) })
})
```

De esta manera, creamos una nueva promesa con la posibilidad de resolverse de dos

maneras:

- Ante un evento *listening*, el manejador llama a la función **resolve** con el servidor conectado como argumento. Lo que se pasa como argumento es lo devuelve la promesa si se resuelve con éxito.
- Ante un evento *error*, el manejador ese evento llama a la función reject con el error recibido. Lo que se pasa como argumento es lo que luego es lanzado en forma de error, como resultado fallido de la promesa (y capturado por un eventual catch).

Para mejorar la legibilidad de nuestro código, podemos modularizarlo utilizando una función:

```
function crearServidor(port) {
  const promesaDeServidorConectado = new Promise((resolve, reject) => {
    const server = app.listen(port)
    server.on('listening', () => { resolve(server) })
    server.on('error', error => { reject(error) })
  })
  return promesaDeServidorConectado
}
```

O lo que es lo mismo:

```
function crearServidor(port) {
  const app = express()

  // acá configuro los manejadores de peticiones
  // y demás detalles del servidor

  return new Promise((resolve, reject) => {
    const server = app.listen(port)
    server.on('listening', () => { resolve(server) })
    server.on('error', error => { reject(error) })
  })
}
```

Luego, podemos usarlo utilizando nuestra sintaxis preferida de async/await:

```
try {
  const servidor = await crearServidor(8080)
  // hacer algo con el servidor conectado
} catch (error) {
  console.log(error.message)
```

```
}
```

Apéndice B - Cerrar el servidor mediante instrucciones de código

Si por algún motivo quisieramos cerrar el servidor, pero no estamos frente a la computadora para detener el proceso (usualmente CTRL + C en la terminal) podemos utilizar el método **close** del servidor ya conectado. Este método recibe un callback opcional, que se ejecuta una vez se haya desconectado completamente en forma satisfactoria.

Ejemplo de código

```
try {
  const servidor = await crearServidor(8080)
  // hacer algo con el servidor conectado
  servidor.close(error => {
    if (error) {
      // no se pudo cerrar!
    } else {
      // se cerró bien!
    }
  })
} catch (error) {
  console.log(error.message)
}
```

Si quisieramos, podríamos hacer lo mismo que antes, y encerrar dentro de una promesa el comportamiento del close, para evitar luego tener que lidiar con callbacks:

```
function crearServidor(port) {
  const app = express()
  return new Promise((resolve, reject) => {
    const server = app.listen(port)
    server.on('listening', () => { resolve(server) })
    server.on('error', error => { reject(error) })

    server.cerrar = () => new Promise((resolveClose, rejectClose) => {
      server.close(error => {
      if (error) { rejectClose(error) } else { resolveClose() }
    }

    })
```

```
  })
}
```

Luego entonces, podemos utilizarlo de la siguiente manera:

```
try {
  const servidor = await crearServidor(8080)
  // hacer algo con el servidor conectado
  await servidor.cerrar()
  // en este punto, el servidor está desconectado
} catch (error) {
  // ojo, en este punto puede haber fallado la conexión o la desconexión
  console.log(error.message)
}
```