Desarrollo del lado servidor con NodeJS, Express y MongoDB

Características de Node[1]

En esta lección verás las características principales de la programación en Node.

Gestión de módulos

Un módulo es un conjunto de funciones o código que puede ser importado dentro de Node. Para hacerlo, utilizamos el método require(*nombre_del_modulo*) y lo guardamos en una variable.

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

El ejemplo previo importa el módulo express. Para ejecutarlo debemos hacer:

```
var app = express();
```

Ahora, desde *app* podemos acceder a las propiedades y métodos de express.

También podemos crear nuestros propios módulos para organizar mejor nuestro proyecto. Si no lo hacemos, significa que tendremos todo el código en un único archivo y eso genera mucho desorden.

Para hacerlo, debemos crear en un archivo separado, supongamos que se llame *mate_basica.js* y agregamos las siguientes líneas:

```
exports.producto = function(a, b) { return a * b; };
exports.suma = function(a, b) { return a + b; };
```

[1] La siguiente lección toma notas del siguiente sitio https://developer.mozilla.org/es/docs/Learn/Server-side/Express_Nodejs/Introduction

Exports es el objeto que se va a exportar y por lo tanto, que se va a asignar al momento de hacer un *require('/mate_basica').* En el ejemplo previo estamos exportando producto y suma. Por lo tanto, para usarlo podremos hacer:

```
var mateBasica = require('./mate_basica');
console.log('El producto de 4 x 10 es ' + mateBasica.producto(4,10));
```

También podemos exportar un objeto o una función, para ello debemos exportar directamente el objeto o función al *module.exports*. Veamos los ejemplos:

```
module.exports = {
        producto: function(a, b) { return a * b; },
        suma: function(a, b) { return a + b; }
}
```

O bien,

```
module.exports = function(){
        producto: function(a, b) { return a * b; },
        suma: function(a, b) { return a + b; }
}
```

APIs Asíncronas

En Javascript se suele trabajar con APIs asíncronas. Para entender mejor a qué se refiere el término asincronismo, podemos comenzar a entender su antónimo: el sincronismo. Por ejemplo, el siguiente código es sincrónico:

```
console.log('primero');

console.log('segundo');
```

Podes probarlo en tu consola de desarrollo[2], y ver que los mensajes se muestran en orden. Esto significa que se ejecuta la primera línea de código, y una vez que termina se ejecuta la siguiente.

[2] Desde el menú: seleccione "Consola del navegador" en el submenú "Desarrollador web" en el menú de Firefox (o en el menú Herramientas si muestra la barra de menús o está en OS X)

En cambio, en una API asíncrona, se interpreta la instrucción, se manda a ejecutar y enseguida se pasa a la siguiente instrucción sin esperar que termine la primera.

```
setTimeout(function() {
  console.log('Primero');
  }, 5000);
console.log('Segundo');
```

El código previo escribirá 'Segundo' y 5 segundos más tarde, 'Primero', porque *setTimeout* se ejecuta de forma asíncrona.

Usar APIs asíncronas sin bloqueos es aún más importante en *Node* que en el navegador, porque *Node* es un entorno de ejecución controlado por eventos de un solo hilo. "Un solo hilo" quiere decir que todas las peticiones al servidor son ejecutadas en el mismo hilo (en vez de dividirse en procesos separados). Esto implica que, si alguna de sus funciones llama a métodos sincrónicos que tomen demasiado tiempo en completarse, bloquearán no solo la solicitud actual, sino también cualquier otra petición que esté siendo manejada por tu aplicación web.

Lo importante de una API asíncrona, es tener control y poder actuar cuando termine su ejecución. La forma común de hacerlo es mediante funciones *callbacks,* que son funciones que se ejecutan al finalizar la tarea asíncrona. En el caso del ejemplo, la función que pasamos por parámetro a *setTimeout* actúa como *callback.* Cuando utilizas muchas funciones asíncronas que necesitan que se ejecuten en orden, debes anidar los callbacks y esto lleva a estructuras muy difíciles de interpretar[3] y manejar. Hay varias buenas prácticas que usaremos para evitar este problema, en particular, trabajaremos con *Promises*[4].

[3] http://callbackhell.com/
[4] https://www.promisejs.org/

Rutas

En el ejemplo "Hola Mundo!" en *Express,* de la lección anterior, definimos una función (callback) manejadora de ruta para peticiones HTTP GET a la raíz del sitio ('/').

```
app.get('/', function(req, res) {
  res.send('Hola Mundo!');
});
```

Hay un número de otros métodos de respuesta[5] para finalizar el ciclo de solicitud/respuesta; por ejemplo, podrás llamar a res.json() para enviar una respuesta JSON o res.sendFile() para enviar un archivo.

El objeto que representa una aplicación de *Express*, también posee métodos para definir los manejadores de rutas para el resto de los verbos HTTP: post(), put(), delete(), options(), trace(), copy(), lock(), mkcol(), move(), purge(), propfind(), proppatch(), unlock(), report(), mkactivity(), checkout(), merge(), m-search(), notify(), subscribe(), unsubscribe(), patch(), search(), y connect().

Usualmente es útil agrupar manejadores de rutas para una parte del sitio y accederlos usando un prefijo de ruta en común.

Ejemplo: un sitio con una sección de administración, podría tener todas las rutas relacionadas a dicha sección en un archivo y ser accedidas con el prefijo de ruta */admin/*.

[5] https://expressjs.com/en/guide/routing.html#response-methods

En Express esto se logra usando el objeto express.Router[6]. Por ejemplo, podemos crear nuestra ruta *admin* en un módulo llamado admin.js y, entonces, exportar el objeto Router, como se muestra debajo:

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res) {
  res.send('Inicio de Admin');
});


router.get('/usuarios', function(req, res) {
  res.send('Usuarios en el sistema');
});

module.exports = router;
```

Para usar el router en nuestro archivo app principal, necesitamos el require() del módulo de rutas admin.js y, luego, enviar el mensaje use() a la aplicación *Express* para agregar el Router al software intermediario (*middleware*) que maneja las rutas. Las dos rutas serán accesibles desde /admin/ y /admin/usuarios/.

```
var admin = require('./admin.js');
app.use('/admin', admin);
```

Uso del middleware

El middleware es ampliamente utilizado en las aplicaciones de Express: desde tareas para entregar archivos estáticos, a la gestión de errores o la compresión de las respuestas HTTP. Mientras las funciones de enrutamiento, con el objeto express.Router, se encargan del ciclo petición-respuesta (al gestionar la respuesta adecuada al cliente), las funciones de middleware normalmente realizan alguna operación al gestionar una petición o respuesta y, a continuación, llaman a la

[6] https://expressjs.com/en/guide/routing.html#express-router

siguiente función en la pila, que puede ser otra función de middleware u otra función de enrutamiento. El orden en el que las funciones de middleware son llamadas depende del desarrollador de la aplicación.

El middleware puede realizar cualquier operación, hacer cambios a una petición, ejecutar código, realizar cambios a la petición o al objeto pedido, puede también finalizar el ciclo de petición-respuesta. Si no finaliza el ciclo, debe llamar a la función next() para pasar el control de la ejecución a la siguiente función del middleware (o la petición quedaría esperando una respuesta...).

La mayoría de las aplicaciones usan middlewares desarrollados por terceras personas, para simplificar funciones habituales en el desarrollo web, como puede ser: gestión de cookies, sesiones, autentificación de usuarios, peticiones POST y datos en JSON, registros de eventos, etc.

Para usar estas colecciones, primero debemos instalar la librería, usando NPM. Veamos un ejemplo utilizando el middleware *morgan*[7] que nos sirve para loguear lo que hace el servidor.

Primero lo instalamos, npm install morgan.

Luego lo usamos.

```
var express = require('express');
var logger = require('morgan');
var app = express();
app.use(logger('dev'));
```

Las funciones Middleware y routing son llamadas en el orden que son declaradas. Para algún middleware el orden es importante (por ejemplo, si el middleware de session depende del middleware de cookie, entonces el manejador de cookie tiene que ser

[7] http://expressjs.com/en/resources/middleware/morgan.html

llamado antes). En general, el middleware es llamado antes de configurar las rutas, si no el manejador de rutas no tendrá acceso a la funcionalidad agregada por tu middleware.

También puedes crear tu propio middleware y usarlo de la siguiente manera:

```
var express = require('express');
var app = express();

// tu middleware
var middleware_function = function(req, res, next) {
  // ... hace algo
  next(); // Ejecuta next() para indicarle a Express que continue con el siguiente
middleware
}

// Configura tu middleware con use() para todas las rutas y verbos
app.use(middleware_function);

// Configura tu middleware con use() para todas la ruta una_ruta y todos los
verbos
app.use('/una_ruta', middleware_function);

// Configura tu middleware con use() para todas la ruta una_ruta y el verbo GET
app.get('/', middleware_function);

app.listen(3000);
```