



BELATRIX

Curso Nodejs

Clase 1

Qué es Node.js?

- JS en lado servidor
- I/O sincrónica o asíncrona
- Montado sobre el motor V8
- Arquitectura orientada a eventos
- Manejo de alta concurrencia
- Altamente escalable

¿Qué problema resuelve Node.js?

- Escalabilidad
- Single Thread
- Mantenimiento sencillo
- Servidor centralizado
- Menor replicación de recursos
- Romper el cuello de botella de diferentes maquinas para servir los datos

¿Cómo resuelve esto Node?

- Cambiando la forma en que se realizan conexiones
- No son hilos de SO, sino una ejecución de eventos dentro del proceso del motor node.js
 - Mantiene el servidor ocupado
 - Reduce al mínimo bloqueos E/S
- Node soportaría miles de llamadas concurrentes con un servidor “humilde”

Lo que Node definitivamente no es:

- No es un producto listo para funcionar como Apache o Tomcat
 - En Node.js se debe implementar el servidor
- Diferente concepto sobre módulos
 - Los módulos se incorporan al núcleo del servidor construido

Otras soluciones similares

- Twisted o Tornado de Python
- Perl Object Environment de Perl
- React de PHP
- libevent o libev de C
- EventMachine de Ruby
- vibe.d de D
- Apache MINA, Netty, Akka, Vert.x, Grizzly o Xsocket de Java.

Hello Node

```
160  var http = require('http');
161 ▼ http.createServer(function (req, res) {
162      res.writeHead(200, {'Content-Type': 'text/plain'});
163      res.end("Hello World!");
164  }).listen(8080);
```

Hello Tornado

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

application = tornado.web.Application([
    (r"/", MainHandler),
])

if __name__ == "__main__":
    application.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```


Porqué usar node?

- Uniformidad del lenguaje de programación utilizado
 - agiliza y simplifica mucho el proceso de codificación
- El motor V8 JavaScript
 - intérprete ultra-rápido escrito en C++
 - acceso ultra rápido a propiedades.
 - generación de código maquina mas eficiente
 - mucha eficiencia en la gestión de memoria (garbage collector)

Porqué usar node?

- Superar la barrera de las 10k llamadas concurrentes
- Romper el cuello de botella de E/S
 - No bloquea el recurso y es altamente eficiente en términos de CPU

Entendiendo el problema del bloqueo de E/S

- *Restaurant C-Food vs Node-Food*

Porqué usar node?

- una comunidad activa y creativa abierta a sugerencias.
 - Hay miles de módulos instalables mediante el comando npm (node package manager)
- JS ha demostrado ser uno de los lenguajes de programación dinámicos mas rápidos por el momento. Es un lenguaje muy popular, es multi plataforma

Porque javascript?

- El problema de los 2mb por hilo generado
- Bloqueo en la entrada-salida
- Son seguros los hilos? mala gestión de memoria y del hardware disponible
- Simplificar la codificación del servidor al no tener que generar el código para manejo de hilos

Node.js I/O API

- Es una pequeña colección de módulos que Node.js nos provee de forma nativa para ayudarnos a gestionar la entrada salida del servidor

Node.js I/O API

- Stream
- Fs
- Http

Y muchas otras....

Node.js I/O API - Stream

- Es una interfaz abstracta implementada por muchos objetos de node, utiles para el envio parcial de datos o chunks
- `require('stream')`
- un Stream puede ser Readable, Writable o Duplex

Node.js I/O API - Stream

- Clase Stream.Readable
 - Eventos
 - Readable
 - Data
 - End
 - Close
 - Error
 - Metodos:
 - Read()
 - SetEncoding()
 - Pause()
 - Resume()

Node.js I/O API - Stream

- Clase Stream.Readable
 - Evento 'readable'

```
2  var readable = getReadableStreamComoSea();
3  readable.on('readable', function() {
4      // Datos disponibles para ser leídos y manipulados
5  })
6
```

Node.js I/O API - Stream

- Classe Stream.Readable
 - Evento 'data'

```
8  var readable = getReadableStreamSomehow();
9  readable.on('data', function(chunk) {
10     console.log('got %d bytes of data', chunk.length);
11  })
12
```

Node.js I/O API - Stream

- Clase Stream.Readable
 - Evento 'end'

```
14 readable.on('data', function(chunk) {  
15     console.log('got %d bytes of data', chunk.length);  
16 })  
17 readable.on('end', function() {  
18     console.log('aca ya no habra mas datos.');
```

Node.js I/O API - Stream

- Clase `Stream.Readable`
 - Evento `'close'`: evento disparado cuando el SO cierra el acceso al recurso solicitado, por ejemplo el descriptor deja de existir
 - Evento `'error'`: sucede al obtener codigos de error, por ejemplo recurso no disponible

Node Js IO API – Stream

- Clase Stream.Readable
 - Metodo 'read'
 - readable.Read(size)

```
22 var readable = getReadableStreamSomehow();
23 readable.on('readable', function() {
24     var chunk;
25     while (null !== (chunk = readable.read())) {
26         console.log('got %d bytes of data', chunk.length);
27     }
28 });
29
```

Node.js I/O API - Stream

- Clase `Stream.Readable`
 - Metodo `'setEncoding'`
 - Uso: `readable.setEncoding(encoding)`
 - Ejemplos: `'utf8'`, `'hex'`

Node.js I/O API - Stream

- Clase Stream.Readable
 - Metodo 'pause'
 - Uso: readable.pause()

```
31 var readable = getReadableStreamSomehow();
32 readable.on('data', function(chunk) {
33     console.log('got %d bytes of data', chunk.length);
34     readable.pause();
35     console.log('there will be no more data for 1 second');
36     setTimeout(function() {
37         console.log('now data will start flowing again');
38         readable.resume();
39     }, 1000);
40 })
41
```


Node.js I/O API - Stream

- Clase Stream.Readable
 - Evento 'close': evento disparado cuando el SO cierra el acceso al recurso solicitado, por ejemplo el descriptor deja de existir
 - Evento 'error': sucede al obtener codigos de error, por ejemplo recurso no disponible

Node.js I/O API - Stream

- Clase Stream.Writable:

- Eventos:

- Drain
 - Finish
 - Error
 - Pipe
 - Unpipe

- Metodos:

- Write()
 - End()

Node.js I/O API - Stream

- Evento 'Drain':

```
50 // Write the data to the supplied writable stream IMM times.
51 // Be attentive to back-pressure.
52 function writeOneMillionTimes(writer, data, encoding, callback) {
53     var i = 1000000;
54     write();
55     function write() {
56         var ok = true;
57         do {
58             i -= 1;
59             if (i === 0) {
60                 // last time!
61                 writer.write(data, encoding, callback);
62             } else {
63                 // see if we should continue, or wait
64                 // don't pass the callback, because we're not done yet.
65                 ok = writer.write(data, encoding);
66             }
67         } while (i > 0 && ok);
68         if (i > 0) {
69             // had to stop early!
70             // write some more once it drains
71             writer.once('drain', write);
72         }
73     }
74 }
```

Node.js I/O API - Stream

- Clase Stream.Writable:
 - Evento 'finish'

```
77 var writer = getWritableStreamSomehow();
78 for (var i = 0; i < 100; i++) {
79     writer.write('hello, #' + i + '!\n');
80 }
81 writer.end('this is the end\n');
82 writer.on('finish', function() {
83     console.error('all writes are now complete.');
```

```
84 });
85
```

Node.js I/O API - Stream

- Clase Stream.Writable:
 - Evento 'pipe'

```
87 var writer = getWritableStreamSomehow();
88 var reader = getReadableStreamSomehow();
89 writer.on('pipe', function(src) {
90     // aca podemos decidir que hacer con la fuente de informacion
91 });
92 reader.pipe(writer);
93
```

Node.js I/O API - Stream

- Clase Stream.Writable:
 - Evento 'unpipe'

```
95  var writer = getWritableStreamSomehow();
96  var reader = getReadableStreamSomehow();
97  writer.on('unpipe', function(src) {
98      // aca podemos manejar el cierre del archivo o lo que fuera necesario
99  });
100  reader.pipe(writer);
101  reader.unpipe(writer);
102
```

Node.js I/O API - Stream

- Clase Stream.Duplex: son implementaciones de streams que implementan a su vez Readable y Writable.
- Aunque quedan fuera del alcance de este curso les nombro algunos ejemplos:
 - tcp sockets
 - zlib streams
 - crypto streams

Node.js I/O API - FS

- Modo de uso: `require('fs')`
- Todos sus metodos tienen su forma sincrona y asincrona

Node.js I/O API - FS

- Llamada Asíncrona

```
104  var fs = require('fs');
105  fs.unlink('/tmp/hello', function (err) {
106      if (err) throw err;
107      console.log('successfully deleted /tmp/hello');
108  });
109
```

- Llamada Síncrona

```
111  var fs = require('fs');
112  try {
113      fs.unlinkSync('/tmp/hello')
114      console.log('successfully deleted /tmp/hello');
115  } catch (e) {
116      // decidir que hacer con la excepcion
117  }
```

Node.js I/O API - FS

- el problema con las llamadas asíncronas es que no se puede asegurar la linealidad de la ejecución

```
120 fs.rename('/tmp/hello', '/tmp/world', function (err) {
121     if (err) throw err;
122     console.log('renamed complete');
123 });
124 fs.stat('/tmp/world', function (err, stats) {
125     if (err) throw err;
126     console.log('stats: ' + JSON.stringify(stats));
127 });
128
```

Node.js I/O API - FS

- Solucion:

```
130 fs.rename('/tmp/hello', '/tmp/world', function (err) {
131     if (err) throw err;
132     fs.stat('/tmp/world', function (err, stats) {
133         if (err) throw err;
134         console.log('stats: ' + JSON.stringify(stats));
135     });
136 });
137
```

Node.js I/O API - FS

- Metodos:
 - `fs.rename(oldPath, newPath, callback)`
 - `fs.renameSync(oldPath, newPath)`
 - `fs.chown(path, uid, gid, callback)`
 - `fs.chownSync(path, uid, gid)`
 - `fs.chmod(path, mode, callback)`
 - `fs.chmodSync(path, mode)`
 - `fs.fchmod(fd, mode, callback)`
 - `fs.fchmodSync(fd, mode)`

Node.js I/O API - FS

- Metodos – Continuacion
 - fs.stat(path, callback)

```
139 { dev: 2114,  
140   ino: 48064969,  
141   mode: 33188,  
142   nlink: 1,  
143   uid: 85,  
144   gid: 100,  
145   rdev: 0,  
146   size: 527,  
147   blksize: 4096,  
148   blocks: 8,  
149   atime: Mon, 10 Oct 2011 23:24:11 GMT,  
150   mtime: Mon, 10 Oct 2011 23:24:11 GMT,  
151   ctime: Mon, 10 Oct 2011 23:24:11 GMT }  
152
```

Node.js I/O API - FS

- `-fs.link(srcpath, dstpath, callback)`
- `-fs.symlink(srcpath, dstpath, [type], callback)`
- `-fs.unlink(path, callback)`
- `-fs.rmdir(path, callback)`
- `-fs.mkdir(path, [mode], callback)`
- `-fs.close(fd, callback)`
- `-fs.open(path, flags, [mode], callback)`
- existen muchos métodos mas

Node.js I/O API - HTTP

Modulo capaz de proveernos un cliente y un servidor HTTP soportando muchas de las funcionalidades del protocolo que normalmente son dificiles de implementar.

```
154 ▼ { 'content-length': '123',  
155     'content-type': 'text/plain',  
156     'connection': 'keep-alive',  
157     'accept': '/*/*' }  
158
```

Node.js I/O API - HTTP

- Metodos:
 - `http.createServer([requestListener])`
 - `server.close([callback])`
 - `server.setTimeout(msecs, callback)`
 - `http.get(options, [callback])`
 - `Server.listen(port)`

Node.js I/O API - HTTP

- Eventos de servidor:
 - 'request'
 - 'connection'
 - 'close'
 - 'checkContinue'
 - 'connect'
 - 'upgrade'
 - 'clientError':

Node.js I/O API - HTTP

- Ejercitemos un poco
 - Básico:
 - Crear un servidor web que a cada request responda con “Hello world”
 - Avanzado:
 - Crear un servidor web que busque y sirva archivos html requeridos desde una pagina
 - Se puede utilizar la agenda
 - Usar FS y HTTP

Node.js I/O API - HTTP

- Solución:

```
160  var http = require('http');  
161 ▼ http.createServer(function (req, res) {  
162      res.writeHead(200, {'Content-Type': 'text/plain'});  
163      res.end("Hello World!");  
164  }).listen(8080);
```

Node Js Vs Servlets

