



EBook Gratis

APRENDIZAJE Node.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#node.js

Tabla de contenido

Acerca de	1
Capítulo 1: Empezando con Node.js	2
Observaciones	2
Versiones	2
Examples	6
Hola servidor HTTP mundial	6
Hola línea de comando mundial	7
Instalación y ejecución de Node.js	8
Ejecutando un programa de nodo	8
Implementando su aplicación en línea	9
Depuración de su aplicación NodeJS	9
Depuración nativa	9
Hola mundo con expreso	10
Hola enrutamiento básico mundial	11
TLS Socket: servidor y cliente	12
Cómo crear una clave y un certificado	12
¡Importante!	12
Servidor de socket TLS	13
TLS Socket Client	14
Hola mundo en el REPL	15
Módulos centrales	15
Todos los módulos básicos de un vistazo	16
¡Cómo poner en marcha un servidor web HTTPS básico!	20
Paso 1: Construir una Autoridad de Certificación	20
Paso 2: instale su certificado como certificado raíz	21
Paso 3: Iniciar su servidor de nodo	21
Capítulo 2: Ambiente	23
Examples	23
Accediendo a las variables del entorno	23

Argumentos de la línea de comandos de process.argv	23
Uso de diferentes propiedades / configuración para diferentes entornos como dev, qa, puest.....	24
Cargando las propiedades del entorno desde un "archivo de propiedades"	25
Capítulo 3: Análisis de argumentos de línea de comando	27
Examples	27
Pasando acción (verbo) y valores	27
Pasando interruptores booleanos	27
Capítulo 4: API de CRUD simple basada en REST	28
Examples	28
API REST para CRUD en Express 3+	28
Capítulo 5: Aplicaciones Web Con Express	29
Introducción	29
Sintaxis	29
Parámetros	29
Examples	30
Empezando	30
Enrutamiento básico	30
Obteniendo información de la solicitud	32
Aplicación express modular	33
Ejemplo mas complicado	33
Usando un motor de plantillas	34
Usando un motor de plantillas	34
Ejemplo de plantilla EJS	35
API JSON con ExpressJS	36
Sirviendo archivos estáticos	36
Carpetas multiples	37
Rutas con nombre en estilo Django	37
Manejo de errores	38
Usando middleware y la próxima devolución de llamada	39
Manejo de errores	41
Hook: Cómo ejecutar código antes de cualquier solicitud y después de cualquier resolución	42
Manejo de solicitudes POST	42

Configuración de cookies con cookie-parser.....	43
Middleware personalizado en Express.....	43
Manejo de errores en Express.....	44
Añadiendo middleware.....	44
Hola Mundo.....	45
Capítulo 6: Asegurando aplicaciones Node.js.....	46
Examples.....	46
Prevenición de falsificación de solicitudes entre sitios (CSRF).....	46
SSL / TLS en Node.js.....	47
Utilizando HTTPS.....	48
Configurando un servidor HTTPS.....	48
Paso 1: Construir una Autoridad de Certificación.....	48
Paso 2: instale su certificado como certificado raíz.....	49
Asegurar la aplicación express.js 3.....	49
Capítulo 7: Async / Await.....	51
Introducción.....	51
Examples.....	51
Funciones asíncronas con el manejo de errores Try-Catch.....	51
Comparación entre Promesas y Async / Await.....	52
Progresión de devoluciones de llamada.....	52
Detiene la ejecución en espera.....	53
Capítulo 8: async.js.....	55
Sintaxis.....	55
Examples.....	55
Paralelo: multitarea.....	55
Llame a async.parallel() con un objeto.....	56
Resolviendo múltiples valores.....	56
Serie: mono-tarea independiente.....	57
Llame a async.series() con un objeto.....	58
Cascada: mono-tarea dependiente.....	58
async.times (para manejar el bucle de una manera mejor).....	59
async.each (Para manejar la matriz de datos de manera eficiente).....	59

async.series (Para manejar eventos uno por uno).....	60
Capítulo 9: Autenticación de Windows bajo node.js.....	61
Observaciones.....	61
Examples.....	61
Usando activedirectory.....	61
Instalación.....	61
Uso.....	61
Capítulo 10: Base de datos (MongoDB con Mangosta).....	62
Examples.....	62
Conexión de mangosta.....	62
Modelo.....	62
Insertar datos.....	63
Leer datos.....	63
Capítulo 11: Biblioteca de mangosta.....	65
Examples.....	65
Conéctate a MongoDB utilizando Mongoose.....	65
Guarde datos en MongoDB utilizando las rutas Mongoose y Express.js.....	65
Preparar.....	65
Código.....	66
Uso.....	67
Encuentre datos en MongoDB utilizando las rutas de Mongoose y Express.js.....	67
Preparar.....	67
Código.....	67
Uso.....	69
Encuentre datos en MongoDB usando Mongoose, Express.js Routes y \$ text Operator.....	69
Preparar.....	69
Código.....	70
Uso.....	71
Índices en modelos.....	72
Funciones útiles de la mangosta.....	74
encontrar datos en mongodb usando promesas.....	74

Preparar	74
Código	74
Uso	76
Capítulo 12: Bluebird Promises	77
Examples	77
Convertir la biblioteca de nodeback a Promesas	77
Promesas funcionales	77
Coroutines (Generadores)	77
Eliminación automática de recursos (Promise.using)	78
Ejecutando en serie	78
Capítulo 13: Buen estilo de codificación	79
Observaciones	79
Examples	79
Programa básico de registro	79
Capítulo 14: Carga automática en los cambios	83
Examples	83
Carga automática de cambios en el código fuente usando nodemon	83
Instalando nodemon globalmente	83
Instalando nodemon localmente	83
Usando nodemon	83
Browsersync	83
Visión general	83
Instalación	84
Usuarios de Windows	84
Uso básico	84
Uso avanzado	84
Grunt.js	85
Gulp.js	85
API	85
Capítulo 15: Casos de uso de Node.js	86
Examples	86

Servidor HTTP.....	86
Consola con el símbolo del sistema.....	86
Capítulo 16: Cierre agraciado.....	88
Examples.....	88
Cierre agraciado - SIGTERM.....	88
Capítulo 17: CLI.....	89
Sintaxis.....	89
Examples.....	89
Opciones de línea de comando.....	89
Capítulo 18: Código Node.js para STDIN y STDOUT sin usar ninguna biblioteca.....	93
Introducción.....	93
Examples.....	93
Programa.....	93
Capítulo 19: Comenzando con el perfilado de nodos.....	94
Introducción.....	94
Observaciones.....	94
Examples.....	94
Perfilando una aplicación de nodo simple.....	94
Capítulo 20: Cómo se cargan los módulos.....	97
Examples.....	97
Modo global.....	97
Cargando modulos.....	97
Cargando un módulo de carpeta.....	97
Capítulo 21: Comunicación cliente-servidor.....	99
Examples.....	99
/ w Express, jQuery y Jade.....	99
Capítulo 22: Comunicación socket.io.....	101
Examples.....	101
"¡Hola Mundo!" Con mensajes de socket.....	101
Capítulo 23: Conectarse a MongoDB.....	102
Introducción.....	102

Sintaxis.....	102
Examples.....	102
Ejemplo simple para conectar mongoDB desde Node.JS.....	102
Una forma sencilla de conectar mongoDB con núcleo Node.JS.....	102
Capítulo 24: Conexión Mysql Pool.....	103
Examples.....	103
Usando un grupo de conexiones sin base de datos.....	103
Capítulo 25: Cortar.....	105
Examples.....	105
Añadir nuevas extensiones para requerir ().....	105
Capítulo 26: Creación de una biblioteca Node.js que admita tanto las promesas como las dev.....	106
Introducción.....	106
Examples.....	106
Módulo de ejemplo y programa correspondiente usando Bluebird.....	106
Capítulo 27: Creando API's con Node.js.....	109
Examples.....	109
OBTENER API utilizando Express.....	109
POST API utilizando Express.....	109
Capítulo 28: csv parser en el nodo js.....	111
Introducción.....	111
Examples.....	111
Usando FS para leer en un CSV.....	111
Capítulo 29: Depuración remota en Node.JS.....	112
Examples.....	112
Configuración de ejecución NodeJS.....	112
Configuración de IntelliJ / Webstorm.....	112
Utilice el proxy para la depuración a través del puerto en Linux.....	113
Capítulo 30: Depurando la aplicación Node.js.....	114
Examples.....	114
Core node.js depurador e inspector de nodos.....	114
Usando el depurador de núcleo.....	114

Referencia de comando.....	114
Usando el inspector de nodos incorporado.....	115
Usando inspector de nodos	115
Capítulo 31: Desafíos de rendimiento.....	118
Examples.....	118
Procesando consultas de larga ejecución con Nodo.....	118
Capítulo 32: Desinstalar Node.js.....	122
Examples.....	122
Desinstale completamente Node.js en Mac OSX.....	122
Desinstalar Node.js en Windows.....	122
Capítulo 33: Despliegue de aplicaciones Node.js en producción.....	123
Examples.....	123
Configurando NODE_ENV = "producción".....	123
Banderas de tiempo de ejecución.....	123
Dependencias.....	123
Administrar la aplicación con el administrador de procesos.....	124
Gestor de procesos PM2.....	124
Despliegue utilizando PM2.....	125
Despliegue usando el administrador de procesos.....	126
Forever.....	126
Uso de diferentes propiedades / configuración para diferentes entornos como dev, qa, puest.....	127
Aprovechando los clusters.....	128
Capítulo 34: Despliegue de la aplicación Node.js sin tiempo de inactividad.....	129
Examples.....	129
Despliegue utilizando PM2 sin tiempo de inactividad.....	129
Capítulo 35: Devolución de llamada a la promesa.....	131
Examples.....	131
Prometiendo una devolución de llamada.....	131
Promisificando manualmente una devolución de llamada.....	132
setTimeout promisificado.....	132
Capítulo 36: Diseño API de descanso: Mejores prácticas.....	133

Examples.....	133
Manejo de errores: OBTENER todos los recursos.....	133
Capítulo 37: ECMAScript 2015 (ES6) con Node.js.....	135
Examples.....	135
const / let declaraciones.....	135
Funciones de flecha.....	135
Ejemplo de función de flecha.....	135
desestructuración.....	136
fluir.....	136
Clase ES6.....	137
Capítulo 38: Ejecutando archivos o comandos con procesos hijo.....	138
Sintaxis.....	138
Observaciones.....	138
Examples.....	138
Generando un nuevo proceso para ejecutar un comando.....	138
Generando un shell para ejecutar un comando.....	139
Generando un proceso para ejecutar un ejecutable.....	140
Capítulo 39: Ejecutando node.js como un servicio.....	141
Introducción.....	141
Examples.....	141
Node.js como un sistema de demonio.....	141
Capítulo 40: Emisores de eventos.....	143
Observaciones.....	143
Examples.....	143
HTTP Analytics a través de un emisor de eventos.....	143
Lo esencial.....	144
Obtenga los nombres de los eventos a los que está suscrito.....	145
Obtenga el número de oyentes registrados para escuchar un evento específico.....	145
Capítulo 41: Enrutamiento de solicitudes ajax con Express.JS.....	147
Examples.....	147
Una implementación sencilla de AJAX.....	147
Capítulo 42: Enrutamiento NodeJs.....	149

Introducción.....	149
Observaciones.....	149
Examples.....	149
Enrutamiento de Express Web Server.....	149
Capítulo 43: Entregar HTML o cualquier otro tipo de archivo.....	154
Syntaxis.....	154
Examples.....	154
Entregar HTML en la ruta especificada.....	154
Estructura de la carpeta.....	154
server.js.....	154
Capítulo 44: Enviando un flujo de archivos al cliente.....	156
Examples.....	156
Uso de fs y pipe para transmitir archivos estáticos desde el servidor.....	156
Streaming Utilizando fluent-ffmpeg.....	157
Capítulo 45: Enviar notificación web.....	158
Examples.....	158
Enviar notificación web utilizando GCM (Google Cloud Messaging System).....	158
Capítulo 46: Estructura del proyecto.....	160
Introducción.....	160
Observaciones.....	160
Examples.....	160
Una sencilla aplicación nodejs con MVC y API.....	160
Capítulo 47: Eventloop.....	163
Introducción.....	163
Examples.....	163
Cómo evolucionó el concepto de bucle de eventos.....	163
Eventloop en pseudo código.....	163
Ejemplo de un servidor HTTP de un solo hilo sin bucle de eventos.....	163
Ejemplo de un servidor HTTP multihilo sin bucle de eventos.....	163
Ejemplo de un servidor HTTP con bucle de eventos.....	164
Capítulo 48: Evitar el infierno de devolución de llamada.....	166

Examples.....	166
Módulo asíncrono.....	166
Módulo asíncrono.....	166
Capítulo 49: Exigir()	168
Introducción.....	168
Sintaxis.....	168
Observaciones.....	168
Examples.....	168
A partir del uso require () con una función y archivo.....	168
A partir del uso require () con un paquete NPM.....	169
Capítulo 50: Exportando e importando el módulo en node.js	171
Examples.....	171
Usando un módulo simple en node.js.....	171
Usando Importaciones En ES6.....	172
Exportando con sintaxis ES6.....	173
Capítulo 51: Exportando y consumiendo módulos	174
Observaciones.....	174
Examples.....	174
Cargando y utilizando un módulo.....	174
Creando un módulo hello-world.js.....	175
Invalidando el caché del módulo.....	176
Construyendo tus propios módulos.....	177
Cada módulo inyectado solo una vez.....	178
Módulo cargando desde node_modules.....	178
Carpeta como modulo.....	179
Capítulo 52: Gestión de errores Node.js	181
Introducción.....	181
Examples.....	181
Creando objeto de error.....	181
Error de lanzamiento.....	181
prueba ... atrapa bloque.....	182
Capítulo 53: Gestor de paquetes de hilo	184

Introducción.....	184
Examples.....	184
Instalación de hilo.....	184
Mac OS.....	184
Homebrew.....	184
MacPorts.....	184
Agregando Hilo a su RUTA.....	184
Windows.....	184
Instalador.....	184
Chocolatey.....	184
Linux.....	185
Debian / Ubuntu.....	185
CentOS / Fedora / RHEL.....	185
Arco.....	185
Solus.....	185
Todas las distribuciones.....	186
Método alternativo de instalación.....	186
Script de shell.....	186
Tarball.....	186
Npm.....	186
Instalación posterior.....	186
Creando un paquete básico.....	186
Instalar el paquete con hilo.....	187
Capítulo 54: gruñido.....	188
Observaciones.....	188
Examples.....	188
Introducción a GruntJs.....	188
Instalación de gruntplugins.....	189
Capítulo 55: Guía para principiantes de NodeJS.....	191
Examples.....	191
Hola Mundo !.....	191

Capítulo 56: herrero	192
Examples	192
Construye un blog simple	192
Capítulo 57: Historia de Nodejs	193
Introducción	193
Examples	193
Eventos clave en cada año	193
2009	193
2010	193
2011	193
2012	193
2013	194
2014	194
2015	194
Q1	194
Q2	194
Q3	195
Q4	195
2016	195
Q1	195
Q2	195
Q3	195
Q4	195
Capítulo 58: http	196
Examples	196
servidor http	196
cliente http	197
Capítulo 59: Instalación de Node.js	199
Examples	199
Instala Node.js en Ubuntu	199
Usando el gestor de paquetes apt	199

Usando la última versión específica (ej. LTS 6.x) directamente desde nodesource	199
Instalación de Node.js en Windows	199
Usando el administrador de versiones de nodos (nvm)	200
Instale Node.js From Source con el administrador de paquetes APT	201
Instalando Node.js en Mac usando el administrador de paquetes	201
Homebrew	201
Macports	202
Instalación utilizando el instalador de MacOS X	202
Compruebe si Node está instalado	203
Instalando Node.js en Raspberry PI	203
Instalación con Node Version Manager bajo Fish Shell con Oh My Fish!	203
Instale Node.js desde la fuente en Centos, RHEL y Fedora	204
Instalando Node.js con n	205
Capítulo 60: Integracion de cassandra	206
Examples	206
Hola Mundo	206
Capítulo 61: Integración de mongodb	207
Sintaxis	207
Parámetros	207
Examples	208
Conectarse a MongoDB	208
Método MongoClient Connect()	208
Inserte un documento	208
Método de recogida insertOne()	209
Leer una coleccion	209
Método de recogida find()	210
Actualizar un documento	210
Método de updateOne()	210
Borrar un documento	211
Método de deleteOne()	211
Eliminar múltiples documentos	211

Método de deleteMany()	212
Conexión simple	212
Conexión simple, utilizando promesas	212
Capítulo 62: Integración de MongoDB para Node.js / Express.js	213
Introducción	213
Observaciones	213
Examples	213
Instalación de MongoDB	213
Creando un Modelo de Mangosta	213
Consultar tu base de datos Mongo	214
Capítulo 63: Integración de MySQL	216
Introducción	216
Examples	216
Consultar un objeto de conexión con parámetros	216
Usando un conjunto de conexiones	216
a. Ejecutando múltiples consultas al mismo tiempo	216
segundo. Lograr multi-tenancy en el servidor de bases de datos con diferentes bases de dat	217
Conectarse a MySQL	218
Consultar un objeto de conexión sin parámetros	218
Ejecutar una serie de consultas con una sola conexión de un grupo	218
Devuelve la consulta cuando se produce un error	219
Grupo de conexiones de exportación	219
Capítulo 64: Integración de pasaportes	221
Observaciones	221
Examples	221
Empezando	221
Autenticación local	221
Autenticación de Facebook	223
Autenticación de usuario-contraseña simple	224
Autenticación de Google Passport	225
Capítulo 65: Integración MSSQL	227
Introducción	227

Observaciones.....	227
Examples.....	227
Conectando con SQL vía. mssql npm module.....	227
Capítulo 66: Integración PostgreSQL.....	229
Examples.....	229
Conectarse a PostgreSQL.....	229
Consulta con objeto de conexión.....	229
Capítulo 67: Interactuando con la consola.....	230
Sintaxis.....	230
Examples.....	230
Explotación florestal.....	230
Módulo de consola.....	230
console.log.....	230
console.error.....	230
console.time, console.timeEnd.....	230
Módulo de proceso.....	231
Formateo.....	231
General.....	231
Colores de fuente.....	231
Colores de fondo.....	232
Capítulo 68: Inyección de dependencia.....	233
Examples.....	233
¿Por qué usar la inyección de dependencia.....	233
Capítulo 69: Koa Framework v2.....	234
Examples.....	234
Hola mundo ejemplo.....	234
Manejo de errores utilizando middleware.....	234
Capítulo 70: La comunicación arduino con nodeJs.....	235
Introducción.....	235
Examples.....	235
Comunicación del nodo Js con Arduino a través de serialport.....	235

Codigo js del nodo.....	235
Código arduino.....	236
Empezando.....	236
Capítulo 71: Localización Nodo JS.....	238
Introducción.....	238
Examples.....	238
utilizando el módulo i18n para mantener la localización en la aplicación node js.....	238
Capítulo 72: Lodash.....	240
Introducción.....	240
Examples.....	240
Filtrar una colección.....	240
Capítulo 73: Loopback - Conector basado en REST.....	241
Introducción.....	241
Examples.....	241
Agregar un conector basado en web.....	241
Capítulo 74: Manejo de excepciones.....	243
Examples.....	243
Manejo de excepciones en Node.Js.....	243
Gestión de excepciones no gestionadas.....	245
Manejo silencioso de excepciones.....	245
Volviendo al estado inicial.....	245
Errores y promesas.....	246
Capítulo 75: Manejo de solicitud POST en Node.js.....	247
Observaciones.....	247
Examples.....	247
Ejemplo de servidor node.js que solo maneja solicitudes POST.....	247
Capítulo 76: Mantener una aplicación de nodo constantemente en ejecución.....	249
Examples.....	249
Usa PM2 como administrador de procesos.....	249
Comandos útiles para monitorear el proceso.....	249
Ejecutando y deteniendo un demonio de Forever.....	250

Carrera continua con nohup.....	251
Proceso de gestión con Forever.....	251
Capítulo 77: Marcos de plantillas.....	252
Examples.....	252
Nunjucks.....	252
Capítulo 78: Marcos de pruebas unitarias.....	254
Examples.....	254
Moca síncrona.....	254
Mocha asíncrono (callback).....	254
Mocha asíncrona (Promesa).....	254
Mocha Asíncrono (asíncrono / await).....	254
Capítulo 79: Módulo de cluster.....	256
Sintaxis.....	256
Observaciones.....	256
Examples.....	256
Hola Mundo.....	256
Ejemplo de cluster.....	257
Capítulo 80: Multihilo.....	259
Introducción.....	259
Observaciones.....	259
Examples.....	259
Racimo.....	259
Proceso infantil.....	260
Capítulo 81: N-API.....	262
Introducción.....	262
Examples.....	262
Hola a N-API.....	262
Capítulo 82: Node.js (express.js) con código de ejemplo angular.js.....	264
Introducción.....	264
Examples.....	264
Creando nuestro proyecto.....	264

Ok, pero ¿cómo creamos el proyecto del esqueleto expreso?	264
¿Cómo expreso funciona, brevemente?	265
Instalando Pug y actualizando el motor de plantillas Express.	265
¿Cómo encaja AngularJS en todo esto?	266
Capítulo 83: Node.js Arquitectura y Trabajos Internos	268
Examples	268
Node.js - bajo el capó	268
Node.js - en movimiento	268
Capítulo 84: Node.js con CORS	270
Examples	270
Habilitar CORS en express.js	270
Capítulo 85: Node.JS con ES6	271
Introducción	271
Examples	271
Nodo ES6 Soporte y creación de un proyecto con Babel	271
Usa JS es6 en tu aplicación NodeJS	272
Requisitos previos:	272
Capítulo 86: Node.js con Oracle	275
Examples	275
Conectarse a Oracle DB	275
Consultar un objeto de conexión sin parámetros	275
Usando un módulo local para facilitar la consulta	276
Capítulo 87: Node.js Design Fundamental	278
Examples	278
La filosofía de Node.js	278
Capítulo 88: Node.js Performance	279
Examples	279
Evento de bucle	279
Ejemplo de operación de bloqueo	279
Ejemplo de operación de IO sin bloqueo	279
Consideraciones de rendimiento	280

Aumentar maxSockets.....	280
Lo esencial.....	280
Configurando tu propio agente.....	280
Desactivación total de Socket Pooling.....	281
Escollos.....	281
Habilitar gzip.....	281
Capítulo 89: Node.js v6 Nuevas características y mejoras.....	283
Introducción.....	283
Examples.....	283
Parámetros de función predeterminados.....	283
Parámetros de descanso.....	283
Operador de propagación.....	283
Funciones de flecha.....	284
"esto" en la función de flecha.....	284
Capítulo 90: Node.JS y MongoDB.....	286
Observaciones.....	286
Examples.....	286
Conexión a una base de datos.....	286
Creando nueva colección.....	287
Insertando Documentos.....	287
Leyendo.....	288
Actualizando.....	288
Métodos.....	289
Actualizar().....	289
UpdateOne.....	289
ActualizarMany.....	289
ReplaceOne.....	290
Borrando.....	290
Capítulo 91: NodeJS con Redis.....	292
Observaciones.....	292
Examples.....	292

Empezando.....	292
Almacenamiento de pares clave-valor.....	293
Algunas operaciones más importantes soportadas por node_redis.....	295
Capítulo 92: NodeJS Frameworks.....	297
Examples.....	297
Marcos de Servidor Web.....	297
Exprimir.....	297
Koa.....	297
Marcos de interfaz de línea de comandos.....	297
Comandante.js.....	297
Vorpal.js.....	298
Capítulo 93: Notificaciones push.....	299
Introducción.....	299
Parámetros.....	299
Examples.....	299
Notificación web.....	299
manzana.....	300
Capítulo 94: npm.....	302
Introducción.....	302
Sintaxis.....	302
Parámetros.....	303
Examples.....	304
Instalando paquetes.....	304
Introducción.....	304
Instalando NPM.....	304
Cómo instalar paquetes.....	305
Instalacion de dependencias.....	307
NPM detrás de un servidor proxy.....	308
Alcances y repositorios.....	308
Desinstalar paquetes.....	309
Versiones semánticas básicas.....	309

Configuración de una configuración de paquete.....	310
Publicando un paquete.....	311
Ejecutando scripts.....	312
La eliminación de paquetes extraños.....	312
Listado de paquetes actualmente instalados.....	313
Actualizando npm y paquetes.....	313
Bloqueo de módulos a versiones específicas.....	314
Configuración de paquetes instalados globalmente.....	314
Vinculación de proyectos para una depuración y desarrollo más rápidos.....	315
Texto de ayuda.....	315
Pasos para vincular dependencias de proyectos.....	315
Pasos para vincular una herramienta global.....	315
Problemas que pueden surgir.....	316
Capítulo 95: nvm - Administrador de versiones de nodo.....	317
Observaciones.....	317
Examples.....	317
Instalar NVM.....	317
Compruebe la versión de NVM.....	317
Instalación de una versión específica del nodo.....	317
Usando una versión de nodo ya instalada.....	318
Instala nvm en Mac OSX.....	318
PROCESO DE INSTALACIÓN.....	318
PRUEBA DE QUE NVM FUE INSTALADO CORRECTAMENTE.....	318
Configuración de alias para la versión de nodo.....	319
Ejecute cualquier comando arbitrario en una subshell con la versión deseada del nodo.....	319
Capítulo 96: OAuth 2.0.....	321
Examples.....	321
OAuth 2 con implementación de Redis - grant_type: contraseña.....	321
Espero ayudar!.....	329
Capítulo 97: paquete.json.....	330
Observaciones.....	330

Examples.....	330
Definición básica del proyecto.....	330
Dependencias.....	330
Dependencias.....	331
Guiones.....	331
Scripts predefinidos.....	331
Scripts definidos por el usuario.....	332
Definición extendida del proyecto.....	333
Explorando package.json.....	333
Capítulo 98: pasaporte.js.....	338
Introducción.....	338
Examples.....	338
Ejemplo de LocalStrategy en passport.js.....	338
Capítulo 99: Programación asíncrona.....	340
Introducción.....	340
Sintaxis.....	340
Examples.....	340
Funciones de devolución de llamada.....	340
Funciones de devolución de llamada en JavaScript.....	340
Devolución de llamadas sincrónica.....	340
Devolución de llamadas asíncronas.....	341
Funciones de devolución de llamada en Node.js.....	342
Ejemplo de código.....	343
Manejo asíncrono de errores.....	344
Trata de atraparlo.....	344
Posibilidades de trabajo.....	344
Controladores de eventos.....	344
Dominios.....	345
Infierno de devolución de llamada.....	345
Promesas nativas.....	346
Capítulo 100: Programación síncrona vs asíncrona en nodejs.....	348

Examples.....	348
Usando async.....	348
Capítulo 101: Readline.....	349
Sintaxis.....	349
Examples.....	349
Lectura de archivos línea por línea.....	349
Solicitar la entrada del usuario a través de CLI.....	349
Capítulo 102: Ruta-controlador-estructura de servicio para ExpressJS.....	351
Examples.....	351
Estructura de directorios Modelo-Rutas-Controladores-Servicios.....	351
Estructura de código de Model-Routes-Controllers-Services.....	351
usuario.model.js.....	351
usuario.rutas.js.....	351
user.controllers.js.....	352
user.services.js.....	352
Capítulo 103: Sequelize.js.....	353
Examples.....	353
Instalación.....	353
Definiendo modelos.....	354
1. sequelize.define (nombre del modelo, atributos, [opciones]).....	354
2. sequelize.import (ruta).....	354
Capítulo 104: Servidor de nodo sin marco.....	356
Observaciones.....	356
Examples.....	356
Servidor de nodo sin marco.....	356
Superando los problemas de CORS.....	357
Capítulo 105: Sistema de archivos de E / S.....	358
Observaciones.....	358
Examples.....	358
Escribir en un archivo usando writeFile o writeFileSync.....	358
Lectura asincrónica de archivos.....	359

Con codificación.....	359
Sin codificar.....	359
Caminos relativos.....	359
Listado de contenidos del directorio con readdir o readdirSync.....	360
Usando un generador.....	360
Leyendo de un archivo de forma síncrona.....	361
Leyendo una cadena.....	361
Eliminando un archivo usando unlink o unlinkSync.....	361
Leyendo un archivo en un Buffer usando streams.....	362
Compruebe los permisos de un archivo o directorio.....	362
Asíncrono.....	363
Síncrono.....	363
Evitar las condiciones de carrera al crear o utilizar un directorio existente.....	363
Comprobando si existe un archivo o un directorio.....	364
Asíncrono.....	364
Síncrono.....	364
Clonando un archivo usando streams.....	365
Copiando archivos por flujos de flujo.....	365
Cambiendo los contenidos de un archivo de texto.....	366
Determinación del conteo de líneas de un archivo de texto.....	366
app.js.....	366
Leyendo un archivo línea por línea.....	366
app.js.....	366
Capítulo 106: Sockets TCP.....	368
Examples.....	368
Un servidor TCP simple.....	368
Un simple cliente TCP.....	368
Capítulo 107: Subir archivo.....	370
Examples.....	370
Carga de un solo archivo usando multer.....	370
Nota:.....	371

Cómo filtrar la carga por extensión:	371
Usando módulo formidable	371
Capítulo 108: Usando Streams	373
Parámetros	373
Examples	373
Leer datos de TextFile con secuencias	373
Corrientes de tubería	374
Creando tu propio flujo legible / escribible	375
¿Por qué Streams?	375
Capítulo 109: Usando WebSocket con Node.JS	378
Examples	378
Instalación de WebSocket	378
Agregando WebSocket a tus archivos	378
Usando WebSocket's y WebSocket Server's	378
Un ejemplo simple de servidor webSocket	378
Capítulo 110: Uso de Browserfy para resolver el error 'requerido' con los navegadores	380
Examples	380
Ejemplo - archivo.js	380
¿Qué está haciendo este fragmento?	380
Instalar Browserfy	380
Importante	381
Qué significa eso?	381
Capítulo 111: Uso de IISNode para alojar aplicaciones web Node.js en IIS	382
Observaciones	382
Directorio virtual / Aplicación anidada con vistas sin errores	382
Versiones	382
Examples	382
Empezando	382
Requerimientos	382
Ejemplo básico de Hello World usando Express	383
Proyecto Strucure	383

server.js - Aplicación Express	383
Configuración y Web.config	383
Configuración.....	384
IISNode Handler.....	384
Reglas de reescritura de URL.....	384
Uso de un directorio virtual de IIS o una aplicación anidada a través de.....	385
Usando Socket.io con IISNode.....	386
Creditos	388

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [node-js](#)

It is an unofficial and free Node.js ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Node.js.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Node.js

Observaciones

Node.js es un marco de E / S asíncrono basado en eventos, sin bloqueo, que utiliza el motor de JavaScript V8 de Google. Se utiliza para desarrollar aplicaciones que hacen un uso intensivo de la capacidad de ejecutar JavaScript tanto en el cliente como en el lado del servidor y, por lo tanto, se benefician de la reutilización del código y la falta de cambio de contexto. Es de código abierto y multiplataforma. Las aplicaciones Node.js están escritas en JavaScript puro y se pueden ejecutar dentro del entorno Node.js en Windows, Linux, etc.

Versiones

Versión	Fecha de lanzamiento
v8.2.1	2017-07-20
v8.2.0	2017-07-19
v8.1.4	2017-07-11
v8.1.3	2017-06-29
v8.1.2	2017-06-15
v8.1.1	2017-06-13
v8.1.0	2017-06-08
v8.0.0	2017-05-30
v7.10.0	2017-05-02
v7.9.0	2017-04-11
v7.8.0	2017-03-29
v7.7.4	2017-03-21
v7.7.3	2017-03-14
v7.7.2	2017-03-08
v7.7.1	2017-03-02
v7.7.0	2017-02-28

Versión	Fecha de lanzamiento
v7.6.0	2017-02-21
v7.5.0	2017-01-31
v7.4.0	2017-01-04
v7.3.0	2016-12-20
v7.2.1	2016-12-06
v7.2.0	2016-11-22
v7.1.0	2016-11-08
v7.0.0	2016-10-25
v6.11.0	2017-06-06
v6.10.3	2017-05-02
v6.10.2	2017-04-04
v6.10.1	2017-03-21
v6.10.0	2017-02-21
v6.9.5	2017-01-31
v6.9.4	2017-01-05
v6.9.3	2017-01-05
v6.9.2	2016-12-06
v6.9.1	2016-10-19
v6.9.0	2016-10-18
v6.8.1	2016-10-14
v6.8.0	2016-10-12
v6.7.0	2016-09-27
v6.6.0	2016-09-14
v6.5.0	2016-08-26
v6.4.0	2016-08-12

Versión	Fecha de lanzamiento
v6.3.1	2016-07-21
v6.3.0	2016-07-06
v6.2.2	2016-06-16
v6.2.1	2016-06-02
v6.2.0	2016-05-17
v6.1.0	2016-05-05
v6.0.0	2016-04-26
v5.12.0	2016-06-23
v5.11.1	2016-05-05
v5.11.0	2016-04-21
v5.10.1	2016-04-05
v5.10	2016-04-01
v5.9	2016-03-16
v5.8	2016-03-09
v5.7	2016-02-23
v5.6	2016-02-09
v5.5	2016-01-21
v5.4	2016-01-06
v5.3	2015-12-15
v5.2	2015-12-09
v5.1	2015-11-17
v5.0	2015-10-29
v4.4	2016-03-08
v4.3	2016-02-09
v4.2	2015-10-12

Versión	Fecha de lanzamiento
v4.1	2015-09-17
v4.0	2015-09-08
io.js v3.3	2015-09-02
io.js v3.2	2015-08-25
io.js v3.1	2015-08-19
io.js v3.0	2015-08-04
io.js v2.5	2015-07-28
io.js v2.4	2015-07-17
io.js v2.3	2015-06-13
io.js v2.2	2015-06-01
io.js v2.1	2015-05-24
io.js v2.0	2015-05-04
io.js v1.8	2015-04-21
io.js v1.7	2015-04-17
io.js v1.6	2015-03-20
io.js v1.5	2015-03-06
io.js v1.4	2015-02-27
io.js v1.3	2015-02-20
io.js v1.2	2015-02-11
io.js v1.1	2015-02-03
io.js v1.0	2015-01-14
v0.12	2016-02-09
v0.11	2013-03-28
v0.10	2013-03-11
v0.9	2012-07-20

Versión	Fecha de lanzamiento
v0.8	2012-06-22
v0.7	2012-01-17
v0.6	2011-11-04
v0.5	2011-08-26
v0.4	2011-08-26
v0.3	2011-08-26
v0.2	2011-08-26
v0.1	2011-08-26

Examples

Hola servidor HTTP mundial

Primero, [instale Node.js](#) para su plataforma.

En este ejemplo, crearemos un servidor HTTP que escucha en el puerto 1337, que envía `Hello, World!` al navegador. Tenga en cuenta que, en lugar de usar el puerto 1337, puede usar cualquier número de puerto de su elección que no esté actualmente en uso por ningún otro servicio.

El módulo `http` es un **módulo principal de** Node.js (un módulo incluido en la fuente de Node.js, que no requiere la instalación de recursos adicionales). El módulo `http` proporciona la funcionalidad para crear un servidor HTTP utilizando el método `http.createServer()`. Para crear la aplicación, cree un archivo que contenga el siguiente código JavaScript.

```
const http = require('http'); // Loads the http module

http.createServer((request, response) => {

  // 1. Tell the browser everything is OK (Status code 200), and the data is in plain text
  response.writeHead(200, {
    'Content-Type': 'text/plain'
  });

  // 2. Write the announced text to the body of the page
  response.write('Hello, World!\n');

  // 3. Tell the server that all of the response headers and body have been sent
  response.end();

}).listen(1337); // 4. Tells the server what port to be on
```

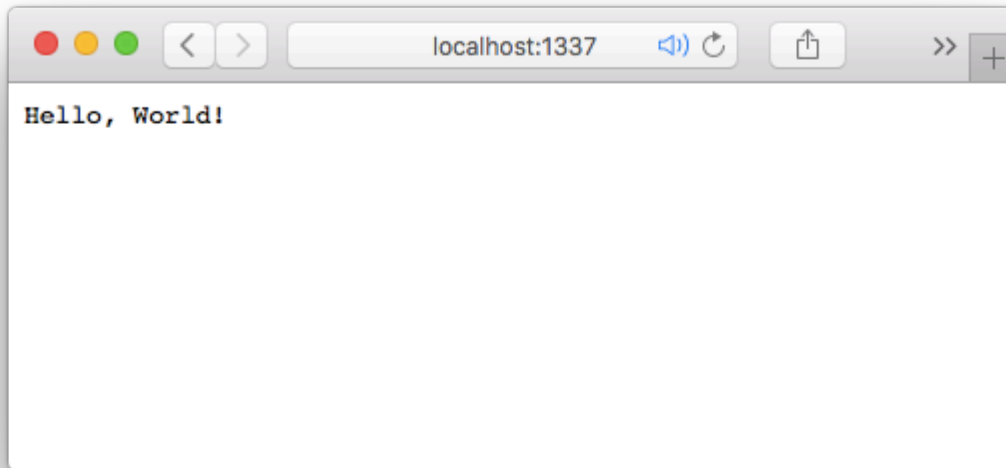
Guarde el archivo con cualquier nombre de archivo. En este caso, si lo llamamos `hello.js` podemos ejecutar la aplicación yendo al directorio donde se encuentra el archivo y usando el

siguiente comando:

```
node hello.js
```

Se puede acceder al servidor creado con la URL <http://localhost:1337> o <http://127.0.0.1:1337> en el navegador.

Aparecerá una página web simple con el texto "¡Hola, mundo!" En la parte superior, como se muestra en la captura de pantalla a continuación.



[Ejemplo editable en línea.](#)

Hola línea de comando mundial

Node.js también se puede utilizar para crear utilidades de línea de comandos. El siguiente ejemplo lee el primer argumento de la línea de comando e imprime un mensaje de saludo.

Para ejecutar este código en un sistema Unix:

1. Crea un nuevo archivo y pega el siguiente código. El nombre del archivo es irrelevante.
2. Haga que este archivo sea ejecutable con `chmod 700 FILE_NAME`
3. Ejecuta la aplicación con `./APP_NAME David`

En Windows, realice el paso 1 y ejecútelo con el `node APP_NAME David`

```
#!/usr/bin/env node

'use strict';

/*
  The command line arguments are stored in the `process.argv` array,
  which has the following structure:
```

```

[0] The path of the executable that started the Node.js process
[1] The path to this application
[2-n] the command line arguments

Example: [ '/bin/node', '/path/to/yourscript', 'arg1', 'arg2', ... ]
src: https://nodejs.org/api/process.html#process_process_argv
*/

// Store the first argument as username.
var username = process.argv[2];

// Check if the username hasn't been provided.
if (!username) {

    // Extract the filename
    var appName = process.argv[1].split(require('path').sep).pop();

    // Give the user an example on how to use the app.
    console.error('Missing argument! Example: %s YOUR_NAME', appName);

    // Exit the app (success: 0, error: 1).
    // An error will stop the execution chain. For example:
    //   ./app.js && ls          -> won't execute ls
    //   ./app.js David && ls -> will execute ls
    process.exit(1);
}

// Print the message to the console.
console.log('Hello %s!', username);

```

Instalación y ejecución de Node.js

Para comenzar, instale Node.js en su computadora de desarrollo.

Windows: navegue a la [página de descarga](#) y descargue / ejecute el instalador.

Mac: vaya a la [página de descarga](#) y descargue / ejecute el instalador. Alternativamente, puede instalar Node a través de Homebrew usando `brew install node`. Homebrew es un gestor de paquetes de línea de comandos para Macintosh, y se puede encontrar más información al respecto en el [sitio web de Homebrew](#).

Linux: siga las instrucciones para su distribución en la [página de instalación de la línea de comandos](#).

Ejecutando un programa de nodo

Para ejecutar un programa Node.js, simplemente ejecute `node app.js` o `nodejs app.js`, donde `app.js` es el nombre de archivo del código fuente de su aplicación de nodo. No es necesario que incluya el sufijo `.js` para que Node encuentre la secuencia de comandos que desea ejecutar.

Alternativamente, bajo los sistemas operativos basados en UNIX, un programa Node puede ejecutarse como un script de terminal. Para hacerlo, debe comenzar con un shebang que apunte al intérprete del nodo, como el nodo `#!/usr/bin/env node`. El archivo también debe configurarse

como ejecutable, lo que se puede hacer usando `chmod` . Ahora el script se puede ejecutar directamente desde la línea de comandos.

Implementando su aplicación en línea

Cuando implementa su aplicación en un entorno alojado (específico de Node.js), este entorno generalmente ofrece una variable de entorno `PORT` que puede utilizar para ejecutar su servidor. Cambiar el número de puerto a `process.env.PORT` permite acceder a la aplicación.

Por ejemplo,

```
http.createServer(function(request, response) {  
  // your server code  
}).listen(process.env.PORT);
```

Además, si desea acceder a este sin conexión durante la depuración, puede utilizar esto:

```
http.createServer(function(request, response) {  
  // your server code  
}).listen(process.env.PORT || 3000);
```

donde `3000` es el número de puerto fuera de línea.

Depuración de su aplicación NodeJS

Puedes usar el inspector de nodos. Ejecute este comando para instalarlo a través de npm:

```
npm install -g node-inspector
```

Entonces puedes depurar tu aplicación usando

```
node-debug app.js
```

El repositorio de Github se puede encontrar aquí: <https://github.com/node-inspector/node-inspector>

Depuración nativa

También puede depurar node.js de forma nativa al iniciarlo así:

```
node debug your-script.js
```

Para interrumpir su depurador exactamente en la línea de código que desea, use esto:

```
debugger;
```

Para más información ver [aquí](#) .

En node.js 8 usa el siguiente comando:

```
node --inspect-brk your-script.js
```

Luego, abra `about://inspect` en una versión reciente de Google Chrome y seleccione su secuencia de comandos Node para obtener la experiencia de depuración de las herramientas de desarrollo de Chrome.

Hola mundo con expreso

El siguiente ejemplo utiliza Express para crear un servidor HTTP que escucha en el puerto 3000, que responde con "¡Hola, mundo!". Express es un marco web de uso común que es útil para crear API de HTTP.

Primero, crea una nueva carpeta, por ejemplo, `myApp` . Vaya a `myApp` y `myApp` un nuevo archivo JavaScript que contenga el siguiente código (llamémoslo `hello.js` por ejemplo). Luego instale el módulo `express` utilizando `npm install --save express` desde la línea de comandos. Consulte [esta documentación](#) para obtener más información sobre cómo instalar paquetes .

```
// Import the top-level function of express
const express = require('express');

// Creates an Express application using the top-level function
const app = express();

// Define port number as 3000
const port = 3000;

// Routes HTTP GET requests to the specified path "/" with the specified callback function
app.get('/', function(request, response) {
  response.send('Hello, World!');
});

// Make the app listen on port 3000
app.listen(port, function() {
  console.log('Server listening on http://localhost:' + port);
});
```

Desde la línea de comandos, ejecute el siguiente comando:

```
node hello.js
```

Abra su navegador y navegue a `http://localhost:3000` o `http://127.0.0.1:3000` para ver la respuesta.

Para obtener más información sobre el marco Express, puede consultar la sección [Aplicaciones web con Express](#)

Hola enrutamiento básico mundial

Una vez que entienda cómo crear un [servidor HTTP](#) con `nodo`, es importante entender cómo hacer que "haga" las cosas en función de la ruta a la que un usuario ha navegado. Este fenómeno se llama, "enrutamiento".

El ejemplo más básico de esto sería verificar `if (request.url === 'some/path/here')` , y luego llamar a una función que responde con un nuevo archivo.

Un ejemplo de esto se puede ver aquí:

```
const http = require('http');

function index (request, response) {
  response.writeHead(200);
  response.end('Hello, World!');
}

http.createServer(function (request, response) {

  if (request.url === '/') {
    return index(request, response);
  }

  response.writeHead(404);
  response.end(http.STATUS_CODES[404]);

}).listen(1337);
```

Sin embargo, si continúa definiendo sus "rutas" de esta manera, terminará con una función de devolución de llamada masiva, y no queremos un lío gigante como ese, así que veamos si podemos limpiar esto.

Primero, almacenemos todas nuestras rutas en un objeto:

```
var routes = {
  '/': function index (request, response) {
    response.writeHead(200);
    response.end('Hello, World!');
  },
  '/foo': function foo (request, response) {
    response.writeHead(200);
    response.end('You are now viewing "foo"');
  }
}
```

Ahora que hemos almacenado 2 rutas en un objeto, ahora podemos verificarlas en nuestra devolución de llamada principal:

```
http.createServer(function (request, response) {

  if (request.url in routes) {
    return routes[request.url](request, response);
  }

}
```

```
response.writeHead(404);  
response.end(http.STATUS_CODES[404]);  
  
}).listen(1337);
```

Ahora, cada vez que intente navegar por su sitio web, comprobará la existencia de esa ruta en sus rutas y llamará a la función correspondiente. Si no se encuentra una ruta, el servidor responderá con un 404 (No encontrado).

Y ahí lo tienen, el enrutamiento con la API del servidor HTTP es muy simple.

TLS Socket: servidor y cliente

Las únicas diferencias importantes entre esto y una conexión TCP regular son la clave privada y el certificado público que deberá establecer en un objeto de opción.

Cómo crear una clave y un certificado

El primer paso en este proceso de seguridad es la creación de una clave privada. ¿Y cuál es esta clave privada? Básicamente, es un conjunto de ruido aleatorio que se utiliza para cifrar la información. En teoría, podría crear una clave y usarla para cifrar lo que quiera. Pero es una buena práctica tener diferentes claves para cosas específicas. Porque si alguien roba su clave privada, es similar a que alguien robe las llaves de su casa. Imagínese si usara la misma llave para bloquear su auto, garaje, oficina, etc.

```
openssl genrsa -out private-key.pem 1024
```

Una vez que tengamos nuestra clave privada, podemos crear una CSR (solicitud de firma de certificado), que es nuestra solicitud para que la clave privada esté firmada por una autoridad competente. Es por eso que debes ingresar información relacionada con tu empresa. Esta información será vista por la autoridad firmante y se usará para verificarlo. En nuestro caso, no importa lo que escriba, ya que en el siguiente paso firmaremos nuestro certificado nosotros mismos.

```
openssl req -new -key private-key.pem -out csr.pem
```

Ahora que hemos completado nuestro trabajo de papel, es hora de fingir que somos una autoridad de firma genial.

```
openssl x509 -req -in csr.pem -signkey private-key.pem -out public-cert.pem
```

Ahora que tiene la clave privada y el certificado público, puede establecer una conexión segura entre dos aplicaciones NodeJS. Y, como puede ver en el código de ejemplo, es un proceso muy simple.

¡Importante!

Desde que creamos el certificado público nosotros mismos, con toda honestidad, nuestro certificado es inútil, porque no somos nadie. El servidor NodeJS no confiará en dicho certificado de forma predeterminada, y es por eso que debemos decirle que realmente confíe en nuestro certificado con la siguiente opción `rejectUnauthorized: false`. **Muy importante** : nunca establezca esta variable en verdadero en un entorno de producción.

Servidor de socket TLS

```
'use strict';

var tls = require('tls');
var fs = require('fs');

const PORT = 1337;
const HOST = '127.0.0.1'

var options = {
  key: fs.readFileSync('private-key.pem'),
  cert: fs.readFileSync('public-cert.pem')
};

var server = tls.createServer(options, function(socket) {

  // Send a friendly message
  socket.write("I am the server sending you a message.");

  // Print the data that we received
  socket.on('data', function(data) {

    console.log('Received: %s [it is %d bytes long]',
      data.toString().replace(/\n/gm, ""),
      data.length);

  });

  // Let us know when the transmission is over
  socket.on('end', function() {

    console.log('EOT (End Of Transmission)');

  });

});

// Start listening on a specific port and address
server.listen(PORT, HOST, function() {

  console.log("I'm listening at %s, on port %s", HOST, PORT);

});

// When an error occurs, show it.
server.on('error', function(error) {

  console.error(error);

  // Close the connection after the error occurred.
```

```
server.destroy();

});
```

TLS Socket Client

```
'use strict';

var tls = require('tls');
var fs = require('fs');

const PORT = 1337;
const HOST = '127.0.0.1'

// Pass the certs to the server and let it know to process even unauthorized certs.
var options = {
  key: fs.readFileSync('private-key.pem'),
  cert: fs.readFileSync('public-cert.pem'),
  rejectUnauthorized: false
};

var client = tls.connect(PORT, HOST, options, function() {

  // Check if the authorization worked
  if (client.authorized) {
    console.log("Connection authorized by a Certificate Authority.");
  } else {
    console.log("Connection not authorized: " + client.authorizationError)
  }

  // Send a friendly message
  client.write("I am the client sending you a message.");

});

client.on("data", function(data) {

  console.log('Received: %s [it is %d bytes long]',
    data.toString().replace(/\n/gm, ""),
    data.length);

  // Close the connection after receiving the message
  client.end();

});

client.on('close', function() {

  console.log("Connection closed");

});

// When an error occurs, show it.
client.on('error', function(error) {

  console.error(error);

  // Close the connection after the error occurred.
```

```
client.destroy();

});
```

Hola mundo en el REPL

Cuando se llama sin argumentos, Node.js inicia un REPL (Read-Eval-Print-Loop) también conocido como el " *shell de nodo* ".

En un indicador de comando escriba el `node` .

```
$ node
>
```

En el indicador de shell Node `>` escriba "Hello World!"

```
$ node
> "Hello World!"
'Hello World!'
```

Módulos centrales

Node.js es un motor Javascript (el motor V8 de Google para Chrome, escrito en C++) que permite ejecutar Javascript fuera del navegador. Si bien hay numerosas bibliotecas disponibles para ampliar las funcionalidades de Node, el motor viene con un conjunto de *módulos centrales* que implementan funcionalidades básicas.

Actualmente hay 34 módulos centrales incluidos en Node:

```
[ 'assert',
  'buffer',
  'c/c++_addons',
  'child_process',
  'cluster',
  'console',
  'crypto',
  'deprecated_apis',
  'dns',
  'domain',
  'Events',
  'fs',
  'http',
  'https',
  'module',
  'net',
  'os',
  'path',
  'punycode',
  'querystring',
  'readline',
  'repl',
  'stream',
  'string_decoder',
```

```
'timers',  
'tls_(ssl)',  
'tracing',  
'tty',  
'dgram',  
'url',  
'util',  
'v8',  
'vm',  
'zlib' ]
```

Esta lista se obtuvo de la API de documentación del nodo <https://nodejs.org/api/all.html> (archivo JSON: <https://nodejs.org/api/all.json>).

Todos los módulos básicos de un vistazo

afirmar

La `assert` módulo proporciona un conjunto simple de pruebas afirmación de que se pueden utilizar para probar invariantes.

buffer

Antes de la introducción de `TypedArray` en ECMAScript 2015 (ES6), el lenguaje JavaScript no tenía ningún mecanismo para leer o manipular flujos de datos binarios. La clase `Buffer` se introdujo como parte de la API Node.js para hacer posible interactuar con flujos de octetos en el contexto de cosas como flujos de TCP y operaciones del sistema de archivos.

Ahora que `TypedArray` se ha agregado en ES6, la clase `Buffer` implementa la API `Uint8Array` de una manera más optimizada y adecuada para los casos de uso de Node.js.

c / c ++ _ addons

Los complementos de Node.js son objetos compartidos enlazados dinámicamente, escritos en C o C ++, que se pueden cargar en Node.js usando la función `require()`, y se usan como si fueran un módulo ordinario de Node.js. Se usan principalmente para proporcionar una interfaz entre JavaScript que se ejecuta en las bibliotecas Node.js y C / C ++.

child_process

El módulo `child_process` proporciona la capacidad de generar procesos secundarios de manera similar, pero no idéntica, a `popen(3)`.

racimo

Una sola instancia de Node.js se ejecuta en un solo hilo. Para aprovechar los sistemas de múltiples núcleos, el usuario a veces querrá iniciar un clúster de procesos Node.js para manejar la carga. El módulo de clúster le permite crear fácilmente procesos secundarios que comparten todos los puertos del servidor.

console

El módulo de la `console` proporciona una consola de depuración simple que es similar al mecanismo de la consola de JavaScript proporcionado por los navegadores web.

crypto

La `crypto` módulo proporciona la funcionalidad criptográfica que incluye un conjunto de contenedores para el hash de OpenSSL, HMAC, cifra, descifrar, firmar y verificar las funciones.

deprecated_apis

Node.js puede desaprobar las API cuando: (a) se considera que el uso de la API no es seguro, (b) se ha puesto a disposición una API alternativa mejorada, o (c) se esperan cambios en la API en un futuro lanzamiento importante .

dns

El módulo `dns` contiene funciones que pertenecen a dos categorías diferentes:

1. Funciones que utilizan las instalaciones del sistema operativo subyacente para realizar la resolución de nombres y que no necesariamente realizan ninguna comunicación de red. Esta categoría contiene solo una función: `dns.lookup()` .
2. Funciones que se conectan a un servidor DNS real para realizar la resolución de nombres y que *siempre* usan la red para realizar consultas de DNS. Esta categoría contiene todas las funciones en el módulo `dns` *excepto* `dns.lookup()` .

dominio

Este módulo está en desuso . Una vez que se haya finalizado una API de reemplazo, este módulo quedará completamente en desuso. La mayoría de los usuarios finales **no** deberían tener motivos para utilizar este módulo. Los usuarios que absolutamente deben tener la funcionalidad que proporcionan los dominios pueden confiar en ella por el momento, pero deben esperar tener que migrar a una solución diferente en el futuro.

Eventos

Gran parte de la API central de Node.js se basa en una arquitectura asincrónica idiomática basada en eventos, en la que ciertos tipos de objetos (llamados "emisores") emiten periódicamente eventos con nombre que hacen que los objetos de función ("escuchas") sean llamados.

fs

El archivo I / O es proporcionado por envoltorios simples alrededor de las funciones POSIX estándar. Para usar este módulo se `require('fs')` . Todos los métodos tienen formas asíncronas y síncronas.

http

Las interfaces HTTP en Node.js están diseñadas para admitir muchas características del protocolo que tradicionalmente han sido difíciles de usar. En particular, mensajes grandes, posiblemente codificados, trozos. La interfaz tiene cuidado de no amortiguar nunca solicitudes o respuestas completas, ya que el usuario puede transmitir datos.

https

HTTPS es el protocolo HTTP sobre TLS / SSL. En Node.js esto se implementa como un módulo separado.

módulo

Node.js tiene un sencillo sistema de carga de módulos. En Node.js, los archivos y los módulos están en correspondencia uno a uno (cada archivo se trata como un módulo separado).

red

El módulo de `net` proporciona una envoltura de red asíncrona. Contiene funciones para crear servidores y clientes (llamados flujos). Puede incluir este módulo con `require('net');`.

os

El módulo `os` proporciona una serie de métodos de utilidad relacionados con el sistema operativo.

camino

El módulo de `path` proporciona utilidades para trabajar con rutas de archivos y directorios.

punycode

La versión del módulo punycode incluida en Node.js está en desuso .

cadena de consulta

El módulo de `querystring` proporciona utilidades para analizar y formatear cadenas de consulta de URL.

readline

El módulo `readline` proporciona una interfaz para leer datos de una secuencia legible (como `process.stdin`) una línea a la vez.

réplica

El módulo `repl` proporciona una implementación de Read-Eval-Print-Loop (REPL) que está disponible como un programa independiente o se puede incluir en otras aplicaciones.

corriente

Un flujo es una interfaz abstracta para trabajar con datos de transmisión en Node.js. El módulo de `stream` proporciona una API base que facilita la creación de objetos que implementan la interfaz de

flujo.

Hay muchos objetos de flujo proporcionados por Node.js. Por ejemplo, una solicitud a un servidor HTTP y `process.stdout` son instancias de flujo.

string_decoder

El módulo `string_decoder` proporciona una API para decodificar objetos `Buffer` en cadenas de una manera que conserva los caracteres codificados de varios bytes UTF-8 y UTF-16.

temporizadores

El módulo `timer` expone una API global para que las funciones de programación se llamen en algún período de tiempo futuro. Debido a que las funciones del temporizador son globales, no es necesario llamar a `require('timers')` para usar la API.

Las funciones de temporizador dentro de Node.js implementan una API similar a la API de temporizadores provista por los navegadores web, pero usan una implementación interna diferente que se construye alrededor [del Node.js Event Loop](#).

tls_ (ssl)

El módulo `tls` proporciona una implementación de los protocolos de Seguridad de la capa de transporte (TLS) y de la Capa de conexión segura (SSL) que se construye sobre OpenSSL.

rastreo

Trace Event proporciona un mecanismo para centralizar la información de seguimiento generada por V8, Node Core y el código de espacio de usuario.

El rastreo se puede habilitar al pasar la `--trace-events-enabled` cuando se inicia una aplicación Node.js.

tty

El módulo `tty` proporciona las clases `tty.ReadStream` y `tty.WriteStream`. En la mayoría de los casos, no será necesario o posible utilizar este módulo directamente.

dgram

El módulo `dgram` proporciona una implementación de sockets de datagramas UDP.

url

El módulo `url` proporciona utilidades para la resolución y el análisis de URL.

util

El módulo `util` está diseñado principalmente para satisfacer las necesidades de las propias API internas de Node.js. Sin embargo, muchas de las utilidades también son útiles para los desarrolladores de aplicaciones y módulos.

v8

El módulo `v8` expone las API que son específicas de la versión de **V8** integrada en el binario Node.js.

Nota : Las API y la implementación están sujetas a cambios en cualquier momento.

vm

El módulo `vm` proporciona API para compilar y ejecutar código en contextos de máquinas virtuales V8. El código JavaScript puede compilarse y ejecutarse inmediatamente o compilarse, guardarse y ejecutarse más tarde.

Nota : el módulo `vm` no es un mecanismo de seguridad. **No lo use para ejecutar código no confiable** .

zlib

El módulo `zlib` proporciona funcionalidad de compresión implementada usando Gzip y Deflate / Inflate.

¡Cómo poner en marcha un servidor web HTTPS básico!

Una vez que tenga node.js instalado en su sistema, simplemente puede seguir el procedimiento a continuación para que un servidor web básico funcione con soporte tanto para HTTP como para HTTPS.

Paso 1: Construir una Autoridad de Certificación

1. cree la carpeta donde desea almacenar su clave y certificado:

```
mkdir conf
```

2. ir a ese directorio:

```
cd conf
```

3. tome este archivo `ca.cnf` para usarlo como acceso directo de configuración:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

4. crear una nueva autoridad de certificación utilizando esta configuración:

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

5. Ahora que tenemos nuestra autoridad de certificación en `ca-key.pem` y `ca-cert.pem` ,

generemos una clave privada para el servidor:

```
openssl genrsa -out key.pem 4096
```

6. tome este archivo `server.cnf` para usarlo como acceso directo de configuración:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

7. generar la solicitud de firma de certificado utilizando esta configuración:

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

8. firmar la solicitud:

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

Paso 2: instale su certificado como certificado raíz

1. Copie su certificado a la carpeta de sus certificados raíz:

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

2. actualizar la tienda de CA:

```
sudo update-ca-certificates
```

Paso 3: Iniciar su servidor de nodo

Primero, desea crear un archivo `server.js` que contenga su código de servidor real.

La configuración mínima para un servidor HTTPS en Node.js sería algo como esto:

```
var https = require('https');
var fs = require('fs');

var httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

var app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

https.createServer(httpsOptions, app).listen(4433);
```

Si también desea admitir solicitudes http, solo necesita hacer una pequeña modificación:

```
var http = require('http');
var https = require('https');
var fs = require('fs');

var httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

var app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

1. vaya al directorio donde se encuentra su `server.js` :

```
cd /path/to
```

2. ejecuta `server.js` :

```
node server.js
```

Lea Empezando con Node.js en línea: <https://riptutorial.com/es/node-js/topic/340/empezando-con-node-js>

Capítulo 2: Ambiente

Examples

Accediendo a las variables del entorno.

La propiedad `process.env` devuelve un objeto que contiene el entorno de usuario.

Devuelve un objeto como este:

```
{
  TERM: 'xterm-256color',
  SHELL: '/usr/local/bin/bash',
  USER: 'maciej',
  PATH: '~/.bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
  PWD: '/Users/maciej',
  EDITOR: 'vim',
  SHLVL: '1',
  HOME: '/Users/maciej',
  LOGNAME: 'maciej',
  _: '/usr/local/bin/node'
}
```

```
process.env.HOME // '/Users/maciej'
```

Si configura la variable de entorno `FOO` en `foobar`, será accesible con:

```
process.env.FOO // 'foobar'
```

Argumentos de la línea de comandos de `process.argv`

`process.argv` es una matriz que contiene los argumentos de la línea de comandos. El primer elemento será `node`, el segundo elemento será el nombre del archivo JavaScript. Los siguientes elementos serán los argumentos de línea de comando adicionales.

Ejemplo de código:

Suma de salida de todos los argumentos de la línea de comando

`index.js`

```
var sum = 0;
for (i = 2; i < process.argv.length; i++) {
  sum += Number(process.argv[i]);
}

console.log(sum);
```

Ejemplo de uso:

```
node index.js 2 5 6 7
```

La salida será 20

Una breve explicación del código:

Aquí, en el bucle `for (i = 2; i < process.argv.length; i++)` bucle comienza con 2 porque los dos primeros elementos de la matriz `process.argv` **siempre** son `['path/to/node.exe', 'path/to/js/file', ...]`

Conversión a número `Number(process.argv[i])` porque los elementos de la matriz `process.argv` **siempre** son cadenas

Uso de diferentes propiedades / configuración para diferentes entornos como dev, qa, puesta en escena, etc.

Las aplicaciones a gran escala a menudo necesitan propiedades diferentes cuando se ejecutan en diferentes entornos. podemos lograrlo pasando argumentos a la aplicación NodeJs y usando el mismo argumento en el proceso del nodo para cargar un archivo de propiedades del entorno específico.

Supongamos que tenemos dos archivos de propiedades para diferentes entornos.

- dev.json

```
{
  PORT : 3000,
  DB : {
    host : "localhost",
    user : "bob",
    password : "12345"
  }
}
```

- qa.json

```
{
  PORT : 3001,
  DB : {
    host : "where_db_is_hosted",
    user : "bob",
    password : "54321"
  }
}
```

El siguiente código en la aplicación exportará el archivo de propiedad respectivo que queremos usar.

Supongamos que el código está en `environment.js`

```
process.argv.forEach(function (val, index, array) {
  var arg = val.split("=");
  if (arg.length > 0) {
    if (arg[0] === 'env') {
      var env = require('./' + arg[1] + '.json');
      module.exports = env;
    }
  }
});
```

Damos argumentos a la aplicación como sigue.

```
node app.js env=dev
```

Si estamos usando un administrador de procesos como *para siempre*, es tan simple como

```
forever start app.js env=dev
```

Cómo utilizar el archivo de configuración

```
var env= require("environment.js");
```

Cargando las propiedades del entorno desde un "archivo de propiedades"

- Instalar el lector de propiedades:

```
npm install properties-reader --save
```

- Crea un **directorio env** para almacenar tus archivos de propiedades:

```
mkdir env
```

- Crear **ambientes.js** :

```
process.argv.forEach(function (val, index, array) {
  var arg = val.split("=");
  if (arg.length > 0) {
    if (arg[0] === 'env') {
      var env = require('./env/' + arg[1] + '.properties');
      module.exports = env;
    }
  }
});
```

- Ejemplo de archivo de propiedades **development.properties** :

```
# Dev properties
[main]
```

```
# Application port to run the node server
app.port=8080

[database]
# Database connection to mysql
mysql.host=localhost
mysql.port=2500
...
```

- Ejemplo de uso de las propiedades cargadas:

```
var enviornment = require('./environments');
var PropertiesReader = require('properties-reader');
var properties = new PropertiesReader(enviornment);

var someVal = properties.get('main.app.port');
```

- Iniciando el servidor express

```
npm start env=development
```

0

```
npm start env=production
```

Lea Ambiente en línea: <https://riptutorial.com/es/node-js/topic/2340/ambiente>

Capítulo 3: Análisis de argumentos de línea de comando

Examples

Pasando acción (verbo) y valores

```
const options = require("commander");

options
  .option("-v, --verbose", "Be verbose");

options
  .command("convert")
  .alias("c")
  .description("Converts input file to output file")
  .option("-i, --in-file <file_name>", "Input file")
  .option("-o, --out-file <file_name>", "Output file")
  .action(doConvert);

options.parse(process.argv);

if (!options.args.length) options.help();

function doConvert(options){
  //do something with options.inFile and options.outFile
};
```

Pasando interruptores booleanos

```
const options = require("commander");

options
  .option("-v, --verbose")
  .parse(process.argv);

if (options.verbose){
  console.log("Let's make some noise!");
}
```

Lea Análisis de argumentos de línea de comando en línea: <https://riptutorial.com/es/node-js/topic/6174/analisis-de-argumentos-de-linea-de-comando>

Capítulo 4: API de CRUD simple basada en REST

Examples

API REST para CRUD en Express 3+

```
var express = require("express"),
    bodyParser = require("body-parser"),
    server = express();

//body parser for parsing request body
server.use(bodyParser.json());
server.use(bodyParser.urlencoded({ extended: true }));

//temporary store for `item` in memory
var itemStore = [];

//GET all items
server.get('/item', function (req, res) {
  res.json(itemStore);
});

//GET the item with specified id
server.get('/item/:id', function (req, res) {
  res.json(itemStore[req.params.id]);
});

//POST new item
server.post('/item', function (req, res) {
  itemStore.push(req.body);
  res.json(req.body);
});

//PUT edited item in-place of item with specified id
server.put('/item/:id', function (req, res) {
  itemStore[req.params.id] = req.body;
  res.json(req.body);
});

//DELETE item with specified id
server.delete('/item/:id', function (req, res) {
  itemStore.splice(req.params.id, 1);
  res.json(req.body);
});

//START SERVER
server.listen(3000, function () {
  console.log("Server running");
});
```

Lea API de CRUD simple basada en REST en línea: <https://riptutorial.com/es/node-js/topic/5850/api-de-crud-simple-basada-en-rest>

Capítulo 5: Aplicaciones Web Con Express

Introducción

Express es un marco de aplicación web Node.js mínimo y flexible, que proporciona un conjunto robusto de características para crear aplicaciones web.

El sitio web oficial de Express es expressjs.com . La fuente se puede encontrar [en GitHub](#) .

Sintaxis

- `app.get` (ruta [, middleware], devolución de llamada [, devolución de llamada ...])
- `app.put` (ruta [, middleware], devolución de llamada [, devolución de llamada ...])
- `app.post` (ruta [, middleware], devolución de llamada [, devolución de llamada ...])
- aplicación ['eliminar'] (ruta [, middleware], devolución de llamada [, devolución de llamada ...])
- `app.use` (ruta [, middleware], devolución de llamada [, devolución de llamada ...])
- `app.use` (devolución de llamada)

Parámetros

Parámetro	Detalles
<code>path</code>	Especifica la parte de la ruta o la URL que manejará la devolución de llamada dada.
<code>middleware</code>	Una o más funciones que serán llamadas antes de la devolución de llamada. Esencialmente un encadenamiento de múltiples funciones de <code>callback</code> de <code>callback</code> . Útil para un manejo más específico, por ejemplo, autorización o manejo de errores.
<code>callback</code>	Una función que se utilizará para manejar solicitudes a la <code>path</code> especificada. Se llamará como <code>callback(request, response, next)</code> , donde <code>request</code> , <code>response</code> y <code>next</code> se describen a continuación.
<code>request</code> <i>devolución de llamada</i>	Un objeto que encapsula detalles sobre la solicitud HTTP a la que se llama la devolución de llamada para que la maneje.
<code>response</code>	Un objeto que se utiliza para especificar cómo debe responder el servidor a la solicitud.
<code>next</code>	Una devolución de llamada que pasa el control a la siguiente ruta coincidente. Acepta un objeto de error opcional.

Examples

Empezando

Primero deberá crear un directorio, acceder a él en su shell e instalar Express usando [npm](#) ejecutando `npm install express --save`

Cree un archivo y `app.js` nombre `app.js` y agregue el siguiente código que crea un nuevo servidor Express y le agrega un punto final (`/ping`) con el método `app.get` :

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

const app = express();

app.get('/ping', (request, response) => {
  response.send('pong');
});

app.listen(8080, 'localhost');
```

Para ejecutar su script use el siguiente comando en su shell:

```
> node app.js
```

Su aplicación aceptará conexiones en el puerto localhost 8080. Si se omite el argumento del nombre de host para `app.listen` , el servidor aceptará las conexiones en la dirección IP de la máquina, así como en el host local. Si el valor del puerto es 0, el sistema operativo asignará un puerto disponible.

Una vez que se ejecuta el script, puede probarlo en un shell para confirmar que obtiene la respuesta esperada, "pong", desde el servidor:

```
> curl http://localhost:8080/ping
pong
```

También puede abrir un navegador web, navegar a la url [http: // localhost: 8080 / ping](http://localhost:8080/ping) para ver el resultado

Enrutamiento básico

Primero crea una aplicación expresa:

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

Entonces puedes definir rutas como esta:

```
app.get('/someUri', function (req, res, next) {})
```

Esa estructura funciona para todos los métodos HTTP y espera una ruta como primer argumento y un controlador para esa ruta, que recibe los objetos de solicitud y respuesta. Entonces, para los métodos HTTP básicos, estas son las rutas

```
// GET www.domain.com/myPath
app.get('/myPath', function (req, res, next) {})

// POST www.domain.com/myPath
app.post('/myPath', function (req, res, next) {})

// PUT www.domain.com/myPath
app.put('/myPath', function (req, res, next) {})

// DELETE www.domain.com/myPath
app.delete('/myPath', function (req, res, next) {})
```

Puede consultar la lista completa de verbos compatibles [aquí](#) . Si desea definir el mismo comportamiento para una ruta y todos los métodos HTTP, puede usar:

```
app.all('/myPath', function (req, res, next) {})
```

o

```
app.use('/myPath', function (req, res, next) {})
```

o

```
app.use('*', function (req, res, next) {})

// * wildcard will route for all paths
```

Puedes encadenar tus definiciones de ruta para un solo camino

```
app.route('/myPath')
  .get(function (req, res, next) {})
  .post(function (req, res, next) {})
  .put(function (req, res, next) {})
```

También puede agregar funciones a cualquier método HTTP. Se ejecutarán antes de la devolución de llamada final y tomarán los parámetros (req, res, next) como argumentos.

```
// GET www.domain.com/myPath
app.get('/myPath', myFunction, function (req, res, next) {})
```

Sus devoluciones de llamada finales se pueden almacenar en un archivo externo para evitar poner demasiado código en un archivo:

```
// other.js
exports.doSomething = function(req, res, next) { /* do some stuff */};
```

Y luego en el archivo que contiene tus rutas:

```
const other = require('./other.js');
app.get('/someUri', myFunction, other.doSomething);
```

Esto hará que su código sea mucho más limpio.

Obteniendo información de la solicitud

Para obtener información de la url de solicitud (observe que `req` es el objeto de solicitud en la función de controlador de rutas). Considere la definición de esta ruta `/settings/:user_id` y este ejemplo particular `/settings/32135?field=name`

```
// get the full path
req.originalUrl // => /settings/32135?field=name

// get the user_id param
req.params.user_id // => 32135

// get the query value of the field
req.query.field // => 'name'
```

También puede obtener los encabezados de la solicitud, como este

```
req.get('Content-Type')
// "text/plain"
```

Para simplificar la obtención de otra información puede usar middlewares. Por ejemplo, para obtener la información del cuerpo de la solicitud, puede usar el middleware del [analyzer](#) del [cuerpo](#), que transformará el cuerpo de la solicitud sin formato en un formato utilizable.

```
var app = require('express')();
var bodyParser = require('body-parser');

app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-urlencoded
```

Ahora supongamos una petición como esta

```
PUT /settings/32135
{
  "name": "Peter"
}
```

Puedes acceder al nombre publicado así

```
req.body.name
// "Peter"
```

De manera similar, puede acceder a las cookies desde la solicitud, también necesita un

middleware como [el analizador de cookies](#).

```
req.cookies.name
```

Aplicación express modular

Para hacer expreso de aplicaciones web modulares utilizan las fábricas de enrutadores:

Módulo:

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // Router factory
  const router = express.Router();

  router.get('/greet', (req, res, next) => {
    res.end(options.greeting);
  });

  return router;
};
```

Solicitud:

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

express()
  .use('/api/v1/', greetMiddleware({ greeting: 'Hello world' }))
  .listen(8080);
```

Esto hará que su aplicación sea modular, personalizable y su código reutilizable.

Al acceder a `http://<hostname>:8080/api/v1/greet` la salida será `Hello world`

Ejemplo mas complicado

Ejemplo con servicios que muestra ventajas de middleware factory.

Módulo:

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // Router factory
  const router = express.Router();
  // Get controller
  const {service} = options;

  router.get('/greet', (req, res, next) => {
```

```

        res.end(
            service.createGreeting(req.query.name || 'Stranger')
        );
    });

    return router;
};

```

Solicitud:

```

// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

class GreetingService {
    constructor(greeting = 'Hello') {
        this.greeting = greeting;
    }

    createGreeting(name) {
        return `${this.greeting}, ${name}!`;
    }
}

express()
    .use('/api/v1/service1', greetMiddleware({
        service: new GreetingService('Hello'),
    }))
    .use('/api/v1/service2', greetMiddleware({
        service: new GreetingService('Hi'),
    }))
    .listen(8080);

```

Al acceder a `http://<hostname>:8080/api/v1/service1/greet?name=World` la salida será `Hello, World` y acceder a `http://<hostname>:8080/api/v1/service2/greet?name=World` la salida será `Hi, World`.

Usando un motor de plantillas

Usando un motor de plantillas

El siguiente código configurará Jade como motor de plantillas. (Nota: Jade ha sido renombrada a pug partir de diciembre de 2015).

```

const express = require('express'); //Imports the express module
const app = express(); //Creates an instance of the express module

const PORT = 3000; //Randomly chosen port

app.set('view engine', 'jade'); //Sets jade as the View Engine / Template Engine
app.set('views', 'src/views'); //Sets the directory where all the views (.jade files) are stored.

//Creates a Root Route
app.get('/', function(req, res){
    res.render('index'); //renders the index.jade file into html and returns as a response.

```

```

The render function optionally takes the data to pass to the view.
});

//Starts the Express server with a callback
app.listen(PORT, function(err) {
  if (!err) {
    console.log('Server is running at port', PORT);
  } else {
    console.log(JSON.stringify(err));
  }
});

```

De manera similar, también se pueden usar otros motores de plantilla, como los `Handlebars` (`hbs`) o `ejs`. Recuerde a `npm install` el motor de plantillas también. Para `Handlebars` usamos el paquete `hbs`, para `Jade` tenemos un paquete de `jade` y para `EJS`, tenemos un paquete `ejs`.

Ejemplo de plantilla EJS

Con `EJS` (como otras plantillas `Express`), puede ejecutar el código del servidor y acceder a las variables de su servidor desde su `HTML`.

En `EJS` se hace usando `"<%"` como etiqueta de inicio y `"%>"` como etiqueta final, las variables pasadas como parámetros de `render` pueden accederse usando `<%=var_name%>`

Por ejemplo, si tiene una matriz de suministros en su código de servidor puedes recorrerlo usando

```

<h1><%= title %></h1>
<ul>
<% for(var i=0; i<supplies.length; i++) { %>
  <li>
    <a href='supplies/<%= supplies[i] %>'>
      <%= supplies[i] %>
    </a>
  </li>
<% } %>

```

Como puede ver en el ejemplo, cada vez que cambia entre el código del lado del servidor y el `HTML`, debe cerrar la etiqueta `EJS` actual y abrir una nueva más adelante. Aquí queríamos crear `li` dentro del comando `for` así que necesitamos cerrar nuestra etiqueta `EJS`. al final del `for` y crear una nueva etiqueta solo para los corchetes

otro ejemplo

Si queremos que la versión predeterminada de entrada sea una variable del lado del servidor, usamos `<%=`

por ejemplo:

```

Message:<br>
<input type="text" value="<%= message %>" name="message" required>

```

Aquí, la variable de mensaje que se pasa desde el lado del servidor será el valor predeterminado de su entrada, tenga en cuenta que si no pasó la variable de mensaje desde el lado del servidor, `EJS` generará una excepción. Puede pasar parámetros usando `res.render('index', {message: message});` (para el archivo `ejs` llamado `index.ejs`).

En las etiquetas EJS también puede usar `if`, `while` o cualquier otro comando javascript que desee.

API JSON con ExpressJS

```
var express = require('express');
var cors = require('cors'); // Use cors module for enable Cross-origin resource sharing

var app = express();
app.use(cors()); // for all routes

var port = process.env.PORT || 8080;

app.get('/', function(req, res) {
  var info = {
    'string_value': 'StackOverflow',
    'number_value': 8476
  }
  res.json(info);

  // or
  /* res.send(JSON.stringify({
    string_value: 'StackOverflow',
    number_value: 8476
  })); */

  //you can add a status code to the json response
  /* res.status(200).json(info) */
})

app.listen(port, function() {
  console.log('Node.js listening on port ' + port)
})
```

En `http://localhost:8080/` output object

```
{
  string_value: "StackOverflow",
  number_value: 8476
}
```

Sirviendo archivos estáticos

Cuando se crea un servidor web con Express, a menudo se requiere que sirva una combinación de contenido dinámico y archivos estáticos.

Por ejemplo, puede tener `index.html` y `script.js` que son archivos estáticos guardados en el sistema de archivos.

Es común usar una carpeta llamada 'pública' para tener archivos estáticos. En este caso, la estructura de la carpeta puede verse como:

```
project root
```

```
├─ server.js
├─ package.json
└─ public
   ├── index.html
   └─ script.js
```

Esta es la forma de configurar Express para servir archivos estáticos:

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

app.use(express.static('public'));
```

Nota: una vez que la carpeta esté configurada, index.html, script.js y todos los archivos en la carpeta "pública" estarán disponibles en la ruta raíz (no debe especificar `/public/` en la url). Esto se debe a que, Express busca los archivos relativos a la carpeta estática configurada. Puede especificar *el prefijo de la ruta virtual* como se muestra a continuación:

```
app.use('/static', express.static('public'));
```

hará que los recursos estén disponibles bajo el prefijo `/static/`.

Carpetas multiples

Es posible definir múltiples carpetas al mismo tiempo:

```
app.use(express.static('public'));
app.use(express.static('images'));
app.use(express.static('files'));
```

Al servir los recursos Express examinará la carpeta en orden de definición. En el caso de archivos con el mismo nombre, se servirá el de la primera carpeta coincidente.

Rutas con nombre en estilo Django

Un gran problema es que las rutas con nombre valiosas no son compatibles con Express out of the box. La solución es instalar un paquete de terceros compatible, por ejemplo, [express-reverse](https://github.com/jaredhuggett/express-reverse) :

```
npm install express-reverse
```

Conéctalo a tu proyecto:

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

Entonces úsalo como:

```
app.get('test', '/hello', function(req, res) {
  res.end('hello');
});
```

```
});
```

La desventaja de este enfoque es que no puede usar el módulo `route` Express como se muestra en [Uso de enrutador avanzado](#) . La solución es pasar su `app` como un parámetro a su fábrica de enrutadores:

```
require('./middlewares/routing')(app);
```

Y utilízalo como:

```
module.exports = (app) => {  
  app.get('test', '/hello', function(req, res) {  
    res.end('hello');  
  });  
};
```

A partir de ahora, puede resolverlo, cómo definir funciones para combinarlas con espacios de nombres personalizados especificados y apuntar a los controladores apropiados.

Manejo de errores

Manejo básico de errores

De forma predeterminada, Express buscará una vista de 'error' en el directorio `/views` para renderizar. Simplemente cree la vista 'error' y colóquela en el directorio de vistas para manejar los errores. Los errores se escriben con el mensaje de error, el estado y el seguimiento de la pila, por ejemplo:

vistas / error.pug

```
html  
  body  
    h1= message  
    h2= error.status  
    p= error.stack
```

Manejo avanzado de errores

Defina sus funciones de middleware para el manejo de errores al final de la pila de funciones de middleware. Estos tienen cuatro argumentos en lugar de tres (`err`, `req`, `res`, `next`) por ejemplo:

app.js

```
// catch 404 and forward to error handler  
app.use(function(req, res, next) {  
  var err = new Error('Not Found');  
  err.status = 404;  
  
  //pass error to the next matching route.  
  next(err);  
});
```

```
// handle error, print stacktrace
app.use(function(err, req, res, next) {
  res.status(err.status || 500);

  res.render('error', {
    message: err.message,
    error: err
  });
});
```

Puede definir varias funciones de middleware de manejo de errores, tal como lo haría con las funciones de middleware regulares.

Usando middleware y la próxima devolución de llamada

Express pasa la `next` devolución de llamada a cada controlador de ruta y función de middleware que puede usarse para romper la lógica de rutas individuales a través de múltiples controladores.

llamar a `next()` sin argumentos, se indica a Express que continúe con el siguiente middleware o controlador de ruta que coincida. Llamar a `next(err)` con un error activará cualquier middleware de controlador de errores. La llamada `next('route')` pasará por alto cualquier middleware posterior en la ruta actual y saltará a la siguiente ruta coincidente. Esto permite desacoplar la lógica del dominio en componentes reutilizables que son autónomos, más sencillos de probar y más fáciles de mantener y cambiar.

Múltiples rutas coincidentes

Las solicitudes a `/api/foo` o a `/api/bar` ejecutarán el controlador inicial para buscar el miembro y luego pasar el control al controlador real para cada ruta.

```
app.get('/api', function(req, res, next) {
  // Both /api/foo and /api/bar will run this
  lookupMember(function(err, member) {
    if (err) return next(err);
    req.member = member;
    next();
  });
});

app.get('/api/foo', function(req, res, next) {
  // Only /api/foo will run this
  doSomethingWithMember(req.member);
});

app.get('/api/bar', function(req, res, next) {
  // Only /api/bar will run this
  doSomethingDifferentWithMember(req.member);
});
```

Manejador de errores

Los manejadores de errores son middleware con la `function(err, req, res, next)` firma `function(err, req, res, next)`. Se pueden configurar por ruta (por ejemplo, `app.get('/foo',`

`function(err, req, res, next) )` pero, por lo general, un solo controlador de errores que presente una página de error es suficiente.

```
app.get('/foo', function(req, res, next) {
  doSomethingAsync(function(err, data) {
    if (err) return next(err);
    renderPage(data);
  });
});

// In the case that doSomethingAsync return an error, this special
// error handler middleware will be called with the error as the
// first parameter.
app.use(function(err, req, res, next) {
  renderErrorPage(err);
});
```

Middleware

Cada una de las funciones anteriores es en realidad una función de middleware que se ejecuta siempre que una solicitud coincide con la ruta definida, pero cualquier número de funciones de middleware se puede definir en una sola ruta. Esto permite que el middleware se defina en archivos separados y que la lógica común se reutilice a través de múltiples rutas.

```
app.get('/bananas', function(req, res, next) {
  getMember(function(err, member) {
    if (err) return next(err);
    // If there's no member, don't try to look
    // up data. Just go render the page now.
    if (!member) return next('route');
    // Otherwise, call the next middleware and fetch
    // the member's data.
    req.member = member;
    next();
  });
}, function(req, res, next) {
  getMemberData(req.member, function(err, data) {
    if (err) return next(err);
    // If this member has no data, don't bother
    // parsing it. Just go render the page now.
    if (!data) return next('route');
    // Otherwise, call the next middleware and parse
    // the member's data. THEN render the page.
    req.member.data = data;
    next();
  });
}, function(req, res, next) {
  req.member.parsedData = parseMemberData(req.member.data);
  next();
});

app.get('/bananas', function(req, res, next) {
  renderBananas(req.member);
});
```

En este ejemplo, cada función de middleware estaría en su propio archivo o en una variable en otra parte del archivo para que pueda reutilizarse en otras rutas.

Manejo de errores

Los documentos básicos se pueden encontrar [aquí](#)

```
app.get('/path/:id(\\d+)', function (req, res, next) { // please note: "next" is passed
  if (req.params.id == 0) // validate param
    return next(new Error('Id is 0')); // go to first Error handler, see below

  // Catch error on sync operation
  var data;
  try {
    data = JSON.parse('/file.json');
  } catch (err) {
    return next(err);
  }

  // If some critical error then stop application
  if (!data)
    throw new Error('Smth wrong');

  // If you need send extra info to Error handler
  // then send custom error (see Appendix B)
  if (smth)
    next(new MyError('smth wrong', arg1, arg2))

  // Finish request by res.render or res.end
  res.status(200).end('OK');
});

// Be sure: order of app.use have matter
// Error handler
app.use(function(err, req, res, next) {
  if (smth-check, e.g. req.url != 'POST')
    return next(err); // go-to Error handler 2.

  console.log(req.url, err.message);

  if (req.xhr) // if req via ajax then send json else render error-page
    res.json(err);
  else
    res.render('error.html', {error: err.message});
});

// Error handler 2
app.use(function(err, req, res, next) {
  // do smth here e.g. check that error is MyError
  if (err instanceof MyError) {
    console.log(err.message, err.arg1, err.arg2);
  }
  ...
  res.end();
});
```

Apéndice A

```
// "In Express, 404 responses are not the result of an error,
// so the error-handler middleware will not capture them."
// You can change it.
```

```
app.use(function(req, res, next) {
  next(new Error(404));
});
```

apéndice B

```
// How to define custom error
var util = require('util');
...
function MyError(message, arg1, arg2) {
  this.message = message;
  this.arg1 = arg1;
  this.arg2 = arg2;
  Error.captureStackTrace(this, MyError);
}
util.inherits(MyError, Error);
MyError.prototype.name = 'MyError';
```

Hook: Cómo ejecutar código antes de cualquier solicitud y después de cualquier resolución

`app.use()` y `middleware` pueden usarse para "antes" y una combinación de los eventos de [cierre](#) y [finalización](#) pueden usarse para "después".

```
app.use(function (req, res, next) {
  function afterResponse() {
    res.removeListener('finish', afterResponse);
    res.removeListener('close', afterResponse);

    // actions after response
  }
  res.on('finish', afterResponse);
  res.on('close', afterResponse);

  // action before request
  // eventually calling `next()`
  next();
});
...
app.use(app.router);
```

Un ejemplo de esto es el `middleware` del [registrador](#), que se agregará al registro después de la respuesta de forma predeterminada.

Solo asegúrese de que este "middleware" se use antes de `app.router` ya que el orden sí importa.

La publicación original está [aquí](#)

Manejo de solicitudes POST

Al igual que usted maneja las solicitudes de obtención en Express con el método `app.get`, puede usar el método `app.post` para manejar las solicitudes posteriores.

Pero antes de que pueda manejar las solicitudes POST, necesitará usar el middleware `body-parser` del `body-parser`. Simplemente analiza el cuerpo de POST, PUT, DELETE y otras solicitudes.

`Body-Parser` middleware `Body-Parser` analiza el cuerpo de la solicitud y lo convierte en un objeto disponible en `req.body`

```
var bodyParser = require('body-parser');

const express = require('express');

const app = express();

// Parses the body for POST, PUT, DELETE, etc.
app.use(bodyParser.json());

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

app.post('/post-data-here', function(req, res, next){

    console.log(req.body); // req.body contains the parsed body of the request.

});

app.listen(8080, 'localhost');
```

Configuración de cookies con cookie-parser

El siguiente es un ejemplo para configurar y leer cookies utilizando el módulo de [análisis de cookies](#):

```
var express = require('express');
var cookieParser = require('cookie-parser'); // module for parsing cookies
var app = express();
app.use(cookieParser());

app.get('/setcookie', function(req, res){
    // setting cookies
    res.cookie('username', 'john doe', { maxAge: 900000, httpOnly: true });
    return res.send('Cookie has been set');
});

app.get('/getcookie', function(req, res) {
    var username = req.cookies['username'];
    if (username) {
        return res.send(username);
    }

    return res.send('No cookie found');
});

app.listen(3000);
```

Middleware personalizado en Express

En Express, puede definir middlewares que se pueden usar para verificar solicitudes o configurar

algunos encabezados en respuesta.

```
app.use(function(req, res, next){ });    // signature
```

Ejemplo

El siguiente código agrega `user` al objeto de solicitud y pasa el control a la siguiente ruta coincidente.

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

//each request will pass through it
app.use(function(req, res, next){
  req.user = 'testuser';
  next();    // it will pass the control to next matching route
});

app.get('/', function(req, res){
  var user = req.user;
  console.log(user); // testuser
  return res.send(user);
});

app.listen(3000);
```

Manejo de errores en Express

En Express, puede definir un controlador de errores unificado para el manejo de errores ocurridos en la aplicación. Defina entonces el controlador al final de todas las rutas y el código lógico.

Ejemplo

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

//GET /names/john
app.get('/names/:name', function(req, res, next){
  if (req.params.name == 'john'){
    return res.send('Valid Name');
  } else{
    next(new Error('Not valid name'));    //pass to error handler
  }
});

//error handler
app.use(function(err, req, res, next){
  console.log(err.stack);    // e.g., Not valid name
  return res.status(500).send('Internal Server Occured');
});

app.listen(3000);
```

Añadiendo middleware

Las funciones de middleware 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-respuesta de la aplicación.

Las funciones de middleware pueden ejecutar cualquier código, realizar cambios en los objetos de `res` y `req`, finalizar el ciclo de respuesta y llamar al middleware siguiente.

Un ejemplo muy común de middleware es el módulo `cors`. Para agregar soporte CORS, simplemente instálelo, solicítelo y ponga esta línea:

```
app.use(cors());
```

Antes de cualquier enrutador o funciones de enrutamiento.

Hola Mundo

Aquí creamos un servidor básico hello world usando Express. Rutas:

- '/'
- '/wiki'

Y para el descanso le dará "404", es decir, página no encontrada.

```
'use strict';

const port = process.env.PORT || 3000;

var app = require('express')();
app.listen(port);

app.get('/', (req, res) => res.send('HelloWorld!'));
app.get('/wiki', (req, res) => res.send('This is wiki page.'));
app.use((req, res) => res.send('404-PageNotFound'));
```

Nota: Hemos colocado la ruta 404 como la última ruta, ya que Express apila las rutas en orden y las procesa para cada solicitud de forma secuencial.

Lea Aplicaciones Web Con Express en línea: <https://riptutorial.com/es/node-js/topic/483/aplicaciones-web-con-express>

Capítulo 6: Asegurando aplicaciones Node.js

Examples

Prevención de falsificación de solicitudes entre sitios (CSRF)

CSRF es un ataque que obliga al usuario final a ejecutar acciones no deseadas en una aplicación web en la que se autentica actualmente.

Puede suceder porque las cookies se envían con cada solicitud a un sitio web, incluso cuando esas solicitudes provienen de un sitio diferente.

Podemos usar el módulo `csrf` para crear el token csrf y validarlo.

Ejemplo

```
var express = require('express')
var cookieParser = require('cookie-parser') //for cookie parsing
var csrf = require('csrf') //csrf module
var bodyParser = require('body-parser') //for body parsing

// setup route middlewares
var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended: false })

// create express app
var app = express()

// parse cookies
app.use(cookieParser())

app.get('/form', csrfProtection, function(req, res) {
  // generate and pass the csrfToken to the view
  res.render('send', { csrfToken: req.csrfToken() })
})

app.post('/process', parseForm, csrfProtection, function(req, res) {
  res.send('data is being processed')
})
```

Entonces, cuando accedemos a `GET /form`, pasará el token `csrfToken` a la vista.

Ahora, dentro de la vista, establezca el valor `csrfToken` como el valor de un campo de entrada oculto llamado `_csrf`.

por ejemplo, para plantillas de `handlebar`

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="{{csrfToken}}">
  Name: <input type="text" name="name">
  <button type="submit">Submit</button>
</form>
```

por ejemplo, para plantillas de `jade`

```
form(action="/process" method="post")
  input (type="hidden", name="_csrf", value=csrfToken)

  span Name:
    input (type="text", name="name", required=true)
  br

  input (type="submit")
```

por ejemplo, para plantillas `ejs`

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="<%= csrfToken %>">
  Name: <input type="text" name="name">
  <button type="submit">Submit</button>
</form>
```

SSL / TLS en Node.js

Si elige manejar SSL / TLS en su aplicación Node.js, considere que también es responsable de mantener la prevención de ataques SSL / TLS en este momento. En muchas arquitecturas servidor-cliente, SSL / TLS termina en un proxy inverso, tanto para reducir la complejidad de la aplicación como para reducir el alcance de la configuración de seguridad.

Si su aplicación Node.js debe manejar SSL / TLS, se puede proteger cargando los archivos de clave y certificado.

Si su proveedor de certificados requiere una cadena de autoridad de certificados (CA), se puede agregar en la opción `ca` como una matriz. Una cadena con varias entradas en un solo archivo se debe dividir en varios archivos y se debe ingresar en el mismo orden en la matriz, ya que Node.js actualmente no admite varias entradas de CA en un archivo. Se proporciona un ejemplo en el siguiente código para los archivos `1_ca.crt` y `2_ca.crt`. Si la matriz de `ca` es requerida y no está configurada correctamente, los navegadores de los clientes pueden mostrar mensajes que no pudieron verificar la autenticidad del certificado.

Ejemplo

```
const https = require('https');
const fs = require('fs');

const options = {
  key: fs.readFileSync('privatekey.pem'),
  cert: fs.readFileSync('certificate.pem'),
  ca: [fs.readFileSync('1_ca.crt'), fs.readFileSync('2_ca.crt')]
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

Utilizando HTTPS

La configuración mínima para un servidor HTTPS en Node.js sería algo como esto:

```
const https = require('https');
const fs = require('fs');

const httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

const app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

https.createServer(httpsOptions, app).listen(4433);
```

Si también desea admitir solicitudes http, solo necesita hacer una pequeña modificación:

```
const http = require('http');
const https = require('https');
const fs = require('fs');

const httpsOptions = {
  key: fs.readFileSync('path/to/server-key.pem'),
  cert: fs.readFileSync('path/to/server-crt.pem')
};

const app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

Configurando un servidor HTTPS

Una vez que tenga node.js instalado en su sistema, simplemente siga el procedimiento a continuación para que un servidor web básico se ejecute con soporte para HTTP y HTTPS.

Paso 1: Construir una Autoridad de Certificación

1. cree la carpeta donde desea almacenar su clave y certificado:

```
mkdir conf
```

2. ir a ese directorio:

```
cd conf
```

3. tome este archivo `ca.cnf` para usarlo como acceso directo de configuración:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

4. crear una nueva autoridad de certificación utilizando esta configuración:

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

5. Ahora que tenemos nuestra autoridad de certificación en `ca-key.pem` y `ca-cert.pem`, generemos una clave privada para el servidor:

```
openssl genrsa -out key.pem 4096
```

6. tome este archivo `server.cnf` para usarlo como acceso directo de configuración:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

7. generar la solicitud de firma de certificado utilizando esta configuración:

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

8. firmar la solicitud:

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

Paso 2: instale su certificado como certificado raíz

1. Copie su certificado a la carpeta de sus certificados raíz:

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

2. actualizar la tienda de CA:

```
sudo update-ca-certificates
```

Asegurar la aplicación express.js 3

La configuración para realizar una conexión segura utilizando express.js (desde la versión 3):

```
var fs = require('fs');
var http = require('http');
var https = require('https');
var privateKey = fs.readFileSync('sslcert/server.key', 'utf8');
var certificate = fs.readFileSync('sslcert/server.crt', 'utf8');

// Define your key and cert
```

```
var credentials = {key: privateKey, cert: certificate};
var express = require('express');
var app = express();

// your express configuration here

var httpServer = http.createServer(app);
var httpsServer = https.createServer(credentials, app);

// Using port 8080 for http and 8443 for https

httpServer.listen(8080);
httpsServer.listen(8443);
```

De esa manera, proporciona middleware expreso al servidor http / https nativo

Si desea que su aplicación se ejecute en puertos por debajo de 1024, deberá usar el comando `sudo` (no recomendado) o usar un proxy inverso (por ejemplo, nginx, haproxy).

Lea **Asegurando aplicaciones Node.js en línea**: <https://riptutorial.com/es/node-js/topic/3473/asegurando-aplicaciones-node-js>

Capítulo 7: Async / Await

Introducción

Async / await es un conjunto de palabras clave que permite escribir código asíncrono de manera procesal sin tener que depender de devoluciones de llamada (*infierno de devolución de llamada*) o encadenamiento de promesa (`.then().then().then()`).

Esto funciona utilizando la palabra clave `await` para suspender el estado de una función asíncrona, hasta la resolución de una promesa, y usando la palabra clave `async` para declarar tales funciones asíncronas, que devuelven una promesa.

Async / await está disponible desde node.js 8 por defecto o 7 usando la bandera `--harmony-async-await`.

Examples

Funciones asíncronas con el manejo de errores Try-Catch

Una de las mejores características de la sintaxis de async / await es que es posible el estilo de codificación try-catch estándar, como si estuviera escribiendo código síncrono.

```
const myFunc = async (req, res) => {
  try {
    const result = await somePromise();
  } catch (err) {
    // handle errors here
  }
};
```

Aquí hay un ejemplo con Express y promise-mysql:

```
router.get('/flags/:id', async (req, res) => {

  try {

    const connection = await pool.createConnection();

    try {
      const sql = `SELECT f.id, f.width, f.height, f.code, f.filename
                    FROM flags f
                    WHERE f.id = ?
                    LIMIT 1`;
      const flags = await connection.query(sql, req.params.id);
      if (flags.length === 0)
        return res.status(404).send({ message: 'flag not found' });

      return res.send({ flags[0] });

    } finally {
```

```

    pool.releaseConnection(connection);
  }

  } catch (err) {
    // handle errors here
  }
});

```

Comparación entre Promesas y Async / Await

Función mediante promesas:

```

function myAsyncFunction() {
  return aFunctionThatReturnsAPromise()
    // doSomething is a sync function
    .then(result => doSomething(result))
    .catch(handleError);
}

```

Así que aquí es cuando Async / Await entra en acción para limpiar nuestra función:

```

async function myAsyncFunction() {
  let result;

  try {
    result = await aFunctionThatReturnsAPromise();
  } catch (error) {
    handleError(error);
  }

  // doSomething is a sync function
  return doSomething(result);
}

```

Entonces, la palabra clave `async` sería similar a escribir `return new Promise((resolve, reject) => {...})`.

Y `await` similar para obtener el resultado en `then` de devolución de llamada.

Aquí les dejo un gif bastante breve que no dejará ninguna duda en mente después de verlo:

[GIF](#)

Progresión de devoluciones de llamada

Al principio había devoluciones de llamada, y las devoluciones de llamada estaban bien:

```

const getTemperature = (callback) => {
  http.get('www.temperature.com/current', (res) => {
    callback(res.data.temperature)
  })
}

const getAirPollution = (callback) => {

```

```

    http.get('www.pollution.com/current', (res) => {
      callback(res.data.pollution)
    });
  }

  getTemperature(function(temp) {
    getAirPollution(function(pollution) {
      console.log(`the temp is ${temp} and the pollution is ${pollution}.`)
      // The temp is 27 and the pollution is 0.5.
    })
  })
})

```

Pero hubo algunos problemas **realmente frustrantes** con las devoluciones de llamada, por lo que todos empezamos a usar promesas.

```

const getTemperature = () => {
  return new Promise((resolve, reject) => {
    http.get('www.temperature.com/current', (res) => {
      resolve(res.data.temperature)
    })
  })
}

const getAirPollution = () => {
  return new Promise((resolve, reject) => {
    http.get('www.pollution.com/current', (res) => {
      resolve(res.data.pollution)
    })
  })
}

getTemperature()
  .then(temp => console.log(`the temp is ${temp}`))
  .then(() => getAirPollution())
  .then(pollution => console.log(`and the pollution is ${pollution}`))
// the temp is 32
// and the pollution is 0.5

```

Esto fue un poco mejor Finalmente, encontramos `async / await`. Que todavía utiliza promesas bajo el capó.

```

const temp = await getTemperature()
const pollution = await getAirPollution()

```

Detiene la ejecución en espera

Si la promesa no devuelve nada, la tarea asíncrona se puede completar con `await`.

```

try{
  await User.findByIdAndUpdate(user._id, {
    $push: {
      tokens: token
    }
  }).exec()
}catch(e){
  handleError(e)
}

```

```
}
```

Lea Async / Await en línea: <https://riptutorial.com/es/node-js/topic/6729/async---await>

Capítulo 8: `async.js`

Sintaxis

- Cada devolución de llamada debe escribirse con esta sintaxis:
- función de devolución de llamada (err, result [, arg1 [, ...]])
- De esta manera, se ve obligado a devolver el error primero y no puede ignorar el manejo de ellos más adelante. `null` es la convención en ausencia de errores.
- devolución de llamada (null, myResult);
- Sus devoluciones de llamada pueden contener más argumentos que *error* y *resultado*, pero es útil solo para un conjunto específico de funciones (cascada, seq, ...)
- devolución de llamada (null, myResult, myCustomArgument);
- Y, por supuesto, enviar errores. Debe hacerlo y manejar los errores (o al menos registrarlos).
- devolución de llamada (err);

Examples

Paralelo: multitarea

`async.parallel` (tareas, *afterTasksCallback*) ejecutará un conjunto de tareas en paralelo y **esperará el final de todas las tareas** (informadas por la función de llamada de devolución de llamada).

Cuando finalizan las tareas, `async` llama a la devolución de llamada principal con todos los errores y todos los resultados de las tareas.

```
function shortTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfShortTime');
  }, 200);
}

function mediumTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfMediumTime');
  }, 500);
}

function longTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfLongTime');
  }, 1000);
}
```

```

}

async.parallel([
  shortTimeFunction,
  mediumTimeFunction,
  longTimeFunction
],
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});

```

Resultado: ["resultOfShortTime", "resultOfMediumTime", "resultOfLongTime"] .

Llame a `async.parallel()` con un objeto

Puede reemplazar el parámetro de matriz de *tareas* por un objeto. En este caso, los resultados también serán un objeto **con las mismas claves que las tareas** .

Es muy útil para calcular algunas tareas y encontrar fácilmente cada resultado.

```

async.parallel({
  short: shortTimeFunction,
  medium: mediumTimeFunction,
  long: longTimeFunction
},
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});

```

Resultado: {short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"} .

Resolviendo múltiples valores

Cada función paralela se pasa una devolución de llamada. Esta devolución de llamada puede devolver un error como primer argumento o valores de éxito después de eso. Si se pasa una devolución de llamada de varios valores de éxito, estos resultados se devuelven como una matriz.

```

async.parallel({
  short: function shortTimeFunction(callback) {
    setTimeout(function() {
      callback(null, 'resultOfShortTime1', 'resultOfShortTime2');
    }, 200);
  }
});

```

```

    },
    medium: function mediumTimeFunction(callback) {
        setTimeout(function() {
            callback(null, 'resultOfMediumTime1', 'resultOfMeiumTime2');
        }, 500);
    }
},
function(err, results) {
    if (err) {
        return console.error(err);
    }

    console.log(results);
});

```

Resultado:

```

{
  short: ["resultOfShortTime1", "resultOfShortTime2"],
  medium: ["resultOfMediumTime1", "resultOfMediumTime2"]
}

```

Serie: mono-tarea independiente

[async.series \(tarefas, afterTasksCallback\)](#) ejecutará un conjunto de tareas. Cada tarea se ejecuta tras otra . Si una tarea falla, **async** detiene inmediatamente la ejecución y salta a la devolución de llamada principal .

Cuando las tareas se completan correctamente, **async** llama a la devolución de llamada "maestra" con todos los errores y todos los resultados de las tareas.

```

function shortTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfShortTime');
    }, 200);
}

function mediumTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfMediumTime');
    }, 500);
}

function longTimeFunction(callback) {
    setTimeout(function() {
        callback(null, 'resultOfLongTime');
    }, 1000);
}

async.series([
    mediumTimeFunction,
    shortTimeFunction,
    longTimeFunction
],

```

```
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});
```

Resultado: ["resultOfMediumTime", "resultOfShortTime", "resultOfLongTime"] .

Llame a `async.series()` con un objeto

Puede reemplazar el parámetro de matriz de *tareas* por un objeto. En este caso, los resultados también serán un objeto **con las mismas claves que las tareas** .

Es muy útil para calcular algunas tareas y encontrar fácilmente cada resultado.

```
async.series({
  short: shortTimeFunction,
  medium: mediumTimeFunction,
  long: longTimeFunction
},
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(results);
});
```

Resultado: {short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"} .

Cascada: mono-tarea dependiente

[async.waterfall \(tareas, afterTasksCallback\)](#) ejecutará un conjunto de tareas. Cada tarea se ejecuta **después de otra, y el resultado de una tarea se pasa a la siguiente tarea** . Como `async.series()` , si una tarea falla, `async` detiene la ejecución y llama inmediatamente a la devolución de llamada principal.

Cuando las tareas se completan correctamente, `async` llama a la devolución de llamada "maestra" con todos los errores y todos los resultados de las tareas.

```
function getUserRequest(callback) {
  // We simulate the request with a timeout
  setTimeout(function() {
    var userResult = {
      name : 'Aamu'
    };

    callback(null, userResult);
  }, 500);
}
```

```

}

function getUserFriendsRequest(user, callback) {
  // Another request simulate with a timeout
  setTimeout(function() {
    var friendsResult = [];

    if (user.name === "Aamu"){
      friendsResult = [{
        name : 'Alice'
      }, {
        name: 'Bob'
      }];
    }

    callback(null, friendsResult);
  }, 500);
}

async.waterfall([
  getUserRequest,
  getUserFriendsRequest
],
function(err, results) {
  if (err) {
    return console.error(err);
  }

  console.log(JSON.stringify(results));
});

```

Resultado: los `results` contienen el segundo parámetro de devolución de llamada de la última función de la cascada, que es `friendsResult` en ese caso.

async.times (para manejar el bucle de una manera mejor)

Para ejecutar una función dentro de un bucle en Node.js, está bien usar una `for` bucle de bucles cortos. Pero el bucle es largo, usar `for` bucle aumentará el tiempo de procesamiento, lo que podría hacer que el proceso del nodo se bloquee. En tales escenarios, puede usar: **async.times**

```

function recursiveAction(n, callback)
{
  //do whatever want to do repeatedly
  callback(err, result);
}

async.times(5, function(n, next) {
  recursiveAction(n, function(err, result) {
    next(err, result);
  });
}, function(err, results) {
  // we should now have 5 result
});

```

Esto se llama en paralelo. Cuando queramos llamarlo uno a la vez, use: **async.timesSeries**

async.each (Para manejar la matriz de datos de manera eficiente)

Cuando queremos manejar una matriz de datos, es mejor usar **async.each** . Cuando queremos realizar algo con todos los datos y queremos obtener la devolución de llamada final una vez que todo está hecho, entonces este método será útil. Esto se maneja de forma paralela.

```
function createUser(userName, callback)
{
    //create user in db
    callback(null)//or error based on creation
}

var arrayOfData = ['Ritu', 'Sid', 'Tom'];
async.each(arrayOfData, function(eachUserName, callback) {

    // Perform operation on each user.
    console.log('Creating user '+eachUserName);
    //Returning callback is must. Else it wont get the final callback, even if we miss to
    return one callback
    createUser(eachUserName, callback);

}, function(err) {
    //If any of the user creation failed may throw error.
    if( err ) {
        // One of the iterations produced an error.
        // All processing will now stop.
        console.log('unable to create user');
    } else {
        console.log('All user created successfully');
    }
});
```

Para hacer uno a la vez puede usar **async.eachSeries**

async.series (Para manejar eventos uno por uno)

/ En async.series, todas las funciones se ejecutan en serie y las salidas consolidadas de cada función se pasan a la devolución de llamada final. por ejemplo

```
var async = require ('async'); async.series ([function (callback) {console.log ('First Execute ..');
callback (null, 'userPersonalData');}, function (callback) {console.log ('Second Execute ..');
devolución de llamada (null, 'userDependentData');}], función (err, resultado) {console.log
(resultado);});
```

//Salida:

Primera ejecución ... Segunda ejecución ... ['userPersonalData', 'userDependentData'] // resultado

Lea async.js en línea: <https://riptutorial.com/es/node-js/topic/3972/async-js>

Capítulo 9: Autenticación de Windows bajo node.js

Observaciones

Hay varios otros APIS de Active Directory, como [activedirectory2](#) y [adldap](#).

Examples

Usando activedirectory

El siguiente ejemplo está tomado de los documentos completos, disponibles [aquí \(GitHub\)](#) o [aquí \(NPM\)](#).

Instalación

```
npm install --save activedirectory
```

Uso

```
// Initialize
var ActiveDirectory = require('activedirectory');
var config = {
  url: 'ldap://dc.domain.com',
  baseDN: 'dc=domain,dc=com'
};
var ad = new ActiveDirectory(config);
var username = 'john.smith@domain.com';
var password = 'password';
// Authenticate
ad.authenticate(username, password, function(err, auth) {
  if (err) {
    console.log('ERROR: ' + JSON.stringify(err));
    return;
  }
  if (auth) {
    console.log('Authenticated!');
  }
  else {
    console.log('Authentication failed!');
  }
});
```

Lea Autenticación de Windows bajo node.js en línea: <https://riptutorial.com/es/node-js/topic/10612/autenticacion-de-windows-bajo-node-js>

Capítulo 10: Base de datos (MongoDB con Mangosta)

Examples

Conexión de mangosta

¡Asegúrate de tener mongodb corriendo primero! `mongod --dbpath data/`

paquete.json

```
"dependencies": {  
  "mongoose": "^4.5.5",  
}
```

server.js (ECMA 6)

```
import mongoose from 'mongoose';  
  
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');  
const db = mongoose.connection;  
db.on('error', console.error.bind(console, 'DB connection error!'));
```

server.js (ECMA 5.1)

```
var mongoose = require('mongoose');  
  
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');  
var db = mongoose.connection;  
db.on('error', console.error.bind(console, 'DB connection error!'));
```

Modelo

Defina su (s) modelo (s):

app / models / user.js (ECMA 6)

```
import mongoose from 'mongoose';  
  
const userSchema = new mongoose.Schema({  
  name: String,  
  password: String  
});  
  
const User = mongoose.model('User', userSchema);  
  
export default User;
```

aplicación / modelo / usuario.js (ECMA 5.1)

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

var userSchema = new mongoose.Schema({
  name: String,
  password: String
});

var User = mongoose.model('User', userSchema);

module.exports = User
```

Insertar datos

ECMA 6:

```
const user = new User({
  name: 'Stack',
  password: 'Overflow',
}) ;

user.save((err) => {
  if (err) throw err;

  console.log('User saved!');
});
```

ECMA5.1:

```
var user = new User({
  name: 'Stack',
  password: 'Overflow',
}) ;

user.save(function (err) {
  if (err) throw err;

  console.log('User saved!');
});
```

Leer datos

ECMA6:

```
User.findOne({
  name: 'stack'
}, (err, user) => {
  if (err) throw err;

  if (!user) {
    console.log('No user was found');
  } else {
    console.log('User was found');
  }
})
```

```
});
```

ECMA5.1:

```
User.findOne({
  name: 'stack'
}, function (err, user) {
  if (err) throw err;

  if (!user) {
    console.log('No user was found');
  } else {
    console.log('User was found');
  }
});
```

Lea Base de datos (MongoDB con Mangosta) en línea: <https://riptutorial.com/es/node-js/topic/6411/base-de-datos--mongodb-con-mangosta->

Capítulo 11: Biblioteca de mangosta

Examples

Conéctate a MongoDB utilizando Mongoose

Primero, instale Mongoose con:

```
npm install mongoose
```

Luego, agréguelo a `server.js` como dependencias:

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

A continuación, cree el esquema de base de datos y el nombre de la colección:

```
var schemaName = new Schema({  
  request: String,  
  time: Number  
}, {  
  collection: 'collectionName'  
});
```

Crea un modelo y conéctate a la base de datos:

```
var Model = mongoose.model('Model', schemaName);  
mongoose.connect('mongodb://localhost:27017/dbName');
```

A continuación, inicie MongoDB y ejecute `server.js` usando `node server.js`

Para verificar si nos hemos conectado con éxito a la base de datos, podemos usar los eventos `open`, `error` del objeto `mongoose.connection`.

```
var db = mongoose.connection;  
db.on('error', console.error.bind(console, 'connection error:'));  
db.once('open', function() {  
  // we're connected!  
});
```

Guarde datos en MongoDB utilizando las rutas Mongoose y Express.js

Preparar

Primero, instale los paquetes necesarios con:

```
npm install express cors mongoose
```

Código

Luego, agregue dependencias a su archivo `server.js`, cree el esquema de la base de datos y el nombre de la colección, cree un servidor Express.js y conéctese a MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});
```

Ahora agregue las rutas Express.js que usaremos para escribir los datos:

```
app.get('/save/:query', cors(), function(req, res) {
  var query = req.params.query;

  var savedata = new Model({
    'request': query,
    'time': Math.floor(Date.now() / 1000) // Time of save the data in unix timestamp
  }).save(function(err, result) {
    if (err) throw err;

    if(result) {
      res.json(result)
    }
  })
});
```

Aquí la variable de `query` será el parámetro `<query>` de la solicitud HTTP entrante, que se guardará en MongoDB:

```
var savedata = new Model({
  'request': query,
  //...
```

Si se produce un error al intentar escribir en MongoDB, recibirá un mensaje de error en la consola. Si todo tiene éxito, verá los datos guardados en formato JSON en la página.

```
//...
}).save(function(err, result) {
  if (err) throw err;

  if(result) {
    res.json(result)
  }
})
//...
```

Ahora, necesita iniciar MongoDB y ejecutar su archivo `server.js` usando `node server.js`.

Uso

Para usar esto para guardar datos, vaya a la siguiente URL en su navegador:

```
http://localhost:8080/save/<query>
```

Donde `<query>` es la nueva solicitud que desea guardar.

Ejemplo:

```
http://localhost:8080/save/JavaScript%20is%20Awesome
```

Salida en formato JSON:

```
{
  __v: 0,
  request: "JavaScript is Awesome",
  time: 1469411348,
  _id: "57957014b93bc8640f2c78c4"
}
```

Encuentre datos en MongoDB utilizando las rutas de Mongoose y Express.js

Preparar

Primero, instale los paquetes necesarios con:

```
npm install express cors mongoose
```

Código

Luego, agregue dependencias a `server.js` , cree el esquema de la base de datos y el nombre de la colección, cree un servidor `Express.js` y conéctese a `MongoDB`:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});
```

Ahora agregue las rutas `Express.js` que usaremos para consultar los datos:

```
app.get('/find/:query', cors(), function(req, res) {
  var query = req.params.query;

  Model.find({
    'request': query
  }, function(err, result) {
    if (err) throw err;
    if (result) {
      res.json(result)
    } else {
      res.send(JSON.stringify({
        error : 'Error'
      }))
    }
  })
})
```

Suponga que los siguientes documentos están en la colección en el modelo:

```
{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710551
}
{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710555
}
{
  "_id" : ObjectId("578abea0522ad414b8eeb55c"),
```

```
"request" : "JavaScript is Awesome",
"time" : 1468710560
}
```

Y el objetivo es encontrar y mostrar todos los documentos que contienen "JavaScript is Awesome" bajo la tecla "request" .

Para esto, inicie MongoDB y ejecute `server.js` con `node server.js` :

Uso

Para usar esto para encontrar datos, vaya a la siguiente URL en un navegador:

```
http://localhost:8080/find/<query>
```

Donde `<query>` es la consulta de búsqueda.

Ejemplo:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

Salida:

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

Encuentre datos en MongoDB usando Mongoose, Express.js Routes y \$ text Operator

Preparar

Primero, instale los paquetes necesarios con:

```
npm install express cors mongoose
```

Código

Luego, agregue dependencias a `server.js`, cree el esquema de la base de datos y el nombre de la colección, cree un servidor Express.js y conéctese a MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});
```

Ahora agregue las rutas Express.js que usaremos para consultar los datos:

```
app.get('/find/:query', cors(), function(req, res) {
  var query = req.params.query;

  Model.find({
    'request': query
  }, function(err, result) {
    if (err) throw err;
    if (result) {
      res.json(result)
    } else {
      res.send(JSON.stringify({
        error : 'Error'
      })))
    }
  })
});
```

Suponga que los siguientes documentos están en la colección en el modelo:

```
{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710551
}
```

```
{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710555
}
{
  "_id" : ObjectId("578abea0522ad414b8eeb55c"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710560
}
```

Y que el objetivo es encontrar y mostrar todos los documentos que contienen solo la palabra "JavaScript" debajo de la tecla "request" .

Para hacer esto, primero cree un *índice de texto* para "request" en la colección. Para esto, agregue el siguiente código a `server.js` :

```
schemaName.index({ request: 'text' });
```

Y reemplazar:

```
Model.find({
  'request': query
}, function(err, result) {
```

Con:

```
Model.find({
  $text: {
    $search: query
  }
}, function(err, result) {
```

Aquí, estamos usando `$search` operadores `$text` y `$search` MongoDB para encontrar todos los documentos en la colección `collectionName` que contiene al menos una palabra de la consulta de búsqueda especificada.

Uso

Para usar esto para encontrar datos, vaya a la siguiente URL en un navegador:

```
http://localhost:8080/find/<query>
```

Donde `<query>` es la consulta de búsqueda.

Ejemplo:

```
http://localhost:8080/find/JavaScript
```

Salida:

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

Índices en modelos.

MongoDB soporta índices secundarios. En Mongoose, definimos estos índices dentro de nuestro esquema. La definición de índices a nivel de esquema es necesaria cuando necesitamos crear índices compuestos.

Conexión de la mangosta

```
var strConnection = 'mongodb://localhost:27017/dbName';
var db = mongoose.createConnection(strConnection)
```

Creando un esquema básico

```
var Schema = require('mongoose').Schema;
var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  },
  email: {
    type: String,
    required: true
  },
  password: {
    type: String,
    required: true
  },
  created: {
    type: Date,
    default: Date.now
  }
});

var userModel = db.model('users', usersSchema);
```

```
module.exports = userModel;
```

De forma predeterminada, la mangosta agrega dos nuevos campos a nuestro modelo, incluso cuando no están definidos en el modelo. Esos campos son:

__carné de identidad

Mongoose asigna a cada uno de sus esquemas un campo `_id` por defecto si uno no se pasa al constructor de Schema. El tipo asignado es un ObjectId que coincide con el comportamiento predeterminado de MongoDB. Si no desea que se agregue un `_id` a su esquema, puede deshabilitarlo con esta opción.

```
var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  }, {
    _id: false
  });
```

__v o versionKey

VersionKey es una propiedad establecida en cada documento cuando fue creada por primera vez por Mongoose. Este valor de claves contiene la revisión interna del documento. El nombre de esta propiedad de documento es configurable.

Puede deshabilitar fácilmente este campo en la configuración del modelo:

```
var usersSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  }, {
    versionKey: false
  });
```

Índices compuestos

Podemos crear otros índices además de los que crea Mongoose.

```
usersSchema.index({username: 1 });
usersSchema.index({email: 1 });
```

En este caso, nuestro modelo tiene dos índices más, uno para el campo nombre de usuario y otro para el campo de correo electrónico. Pero podemos crear índices compuestos.

```
usersSchema.index({username: 1, email: 1 });
```

Índice de impacto en el rendimiento

De forma predeterminada, la mangosta siempre llama secuencialmente al asegurador para cada índice y emite un evento de "índice" en el modelo cuando todas las llamadas al asegurador tuvieron éxito o cuando hubo un error.

En MongoDB, asegúrese de que el índice esté en desuso desde la versión 3.0.0, ahora es un alias para `createIndex`.

Se recomienda desactivar el comportamiento configurando la opción `autoIndex` de su esquema en falso, o globalmente en la conexión configurando la opción `config.autoIndex` en falso.

```
usersSchema.set('autoIndex', false);
```

Funciones útiles de la mangosta

Mongoose contiene algunas funciones integradas que se basan en el estándar `find()`.

```
doc.find({'some.value':5},function(err,docs){
  //returns array docs
});

doc.findOne({'some.value':5},function(err,doc){
  //returns document doc
});

doc.findById(obj._id,function(err,doc){
  //returns document doc
});
```

encontrar datos en mongodb usando promesas

Preparar

Primero, instale los paquetes necesarios con:

```
npm install express cors mongoose
```

Código

Luego, agregue dependencias a `server.js`, cree el esquema de la base de datos y el nombre de la colección, cree un servidor Express.js y conéctese a MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();
```

```

var schemaName = new Schema({
  request: String,
  time: Number
}, {
  collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
  console.log('Node.js listening on port ' + port);
});

app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});

app.use(function(req, res, next) {
  res.status(404).send('Sorry cant find that!');
});

```

Ahora agregue las rutas Express.js que usaremos para consultar los datos:

```

app.get('/find/:query', cors(), function(req, res, next) {
  var query = req.params.query;

  Model.find({
    'request': query
  })
  .exec() //remember to add exec, queries have a .then attribute but aren't promises
  .then(function(result) {
    if (result) {
      res.json(result)
    } else {
      next() //pass to 404 handler
    }
  })
  .catch(next) //pass to error handler
});

```

Suponga que los siguientes documentos están en la colección en el modelo:

```

{
  "_id" : ObjectId("578abe97522ad414b8eeb55a"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710551
}
{
  "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710555
}
{
  "_id" : ObjectId("578abea0522ad414b8eeb55c"),
  "request" : "JavaScript is Awesome",
  "time" : 1468710560
}

```

```
}
```

Y el objetivo es encontrar y mostrar todos los documentos que contienen "JavaScript is Awesome" bajo la tecla "request" .

Para esto, inicie MongoDB y ejecute `server.js` con `node server.js` :

Uso

Para usar esto para encontrar datos, vaya a la siguiente URL en un navegador:

```
http://localhost:8080/find/<query>
```

Donde `<query>` es la consulta de búsqueda.

Ejemplo:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

Salida:

```
[{
  _id: "578abe97522ad414b8eeb55a",
  request: "JavaScript is Awesome",
  time: 1468710551,
  __v: 0
},
{
  _id: "578abe9b522ad414b8eeb55b",
  request: "JavaScript is Awesome",
  time: 1468710555,
  __v: 0
},
{
  _id: "578abea0522ad414b8eeb55c",
  request: "JavaScript is Awesome",
  time: 1468710560,
  __v: 0
}]
```

Lea Biblioteca de mangosta en línea: <https://riptutorial.com/es/node-js/topic/3486/biblioteca-de-mangosta>

Capítulo 12: Bluebird Promises

Examples

Convertir la biblioteca de nodeback a Promesas

```
const Promise = require('bluebird'),
      fs = require('fs')

Promise.promisifyAll(fs)

// now you can use promise based methods on 'fs' with the Async suffix
fs.readFileAsync('file.txt').then(contents => {
  console.log(contents)
}).catch(err => {
  console.error('error reading', err)
})
```

Promesas funcionales

Ejemplo de mapa:

```
Promise.resolve([ 1, 2, 3 ]).map(el => {
  return Promise.resolve(el * el) // return some async operation in real world
})
```

Ejemplo de filtro:

```
Promise.resolve([ 1, 2, 3 ]).filter(el => {
  return Promise.resolve(el % 2 === 0) // return some async operation in real world
}).then(console.log)
```

Ejemplo de reducir:

```
Promise.resolve([ 1, 2, 3 ]).reduce((prev, curr) => {
  return Promise.resolve(prev + curr) // return some async operation in real world
}).then(console.log)
```

Coroutines (Generadores)

```
const promiseReturningFunction = Promise.coroutine(function* (file) {
  const data = yield fs.readFileAsync(file) // this returns a Promise and resolves to the file contents

  return data.toString().toUpperCase()
})

promiseReturningFunction('file.txt').then(console.log)
```

Eliminación automática de recursos (Promise.using)

```
function somethingThatReturnsADisposableResource() {
  return getSomeResourceAsync(...).disposer(resource => {
    resource.dispose()
  })
}

Promise.using(somethingThatReturnsADisposableResource(), resource => {
  // use the resource here, the disposer will automatically close it when Promise.using exits
})
```

Ejecutando en serie

```
Promise.resolve([1, 2, 3])
  .mapSeries(el => Promise.resolve(el * el)) // in real world, use Promise returning async
function
  .then(console.log)
```

Lea Bluebird Promises en línea: <https://riptutorial.com/es/node-js/topic/6728/bluebird-promises>

Capítulo 13: Buen estilo de codificación

Observaciones

Recomendaría a un principiante comenzar con este estilo de codificación. Y si alguien puede sugerir una mejor manera (ps, opté por esta técnica y está funcionando de manera eficiente para mí en una aplicación utilizada por más de 100k usuarios), no dude en realizar cualquier sugerencia. TIA.

Examples

Programa básico de registro.

A través de este ejemplo, se explicará **cómo** dividir el código **node.js** en diferentes **módulos / carpetas** para una mejor comprensión. Seguir esta técnica hace que sea más fácil para otros desarrolladores entender el código, ya que puede referirse directamente al archivo en cuestión en lugar de revisar todo el código. El uso principal es cuando trabajas en un equipo y un nuevo desarrollador se une en una etapa posterior, le será más fácil combinar con el código en sí.

index.js : - Este archivo gestionará la conexión del servidor.

```
//Import Libraries
var express = require('express'),
    session = require('express-session'),
    mongoose = require('mongoose'),
    request = require('request');

//Import custom modules
var userRoutes = require('./app/routes/userRoutes');
var config = require('./app/config/config');

//Connect to Mongo DB
mongoose.connect(config.getDBString());

//Create a new Express application and Configure it
var app = express();

//Configure Routes
app.use(config.API_PATH, userRoutes());

//Start the server
app.listen(config.PORT);
console.log('Server started at - ' + config.URL + ":" + config.PORT);
```

config.js : -Este archivo administrará todos los parámetros relacionados con la configuración que permanecerán igual durante todo el proceso.

```
var config = {
  VERSION: 1,
  BUILD: 1,
```

```

URL: 'http://127.0.0.1',
API_PATH : '/api',
PORT : process.env.PORT || 8080,
DB : {
    //MongoDB configuration
    HOST : 'localhost',
    PORT : '27017',
    DATABASE : 'db'
},

/*
 * Get DB Connection String for connecting to MongoDB database
 */
getDBString : function(){
    return 'mongodb://' + this.DB.HOST + ':' + this.DB.PORT + '/' + this.DB.DATABASE;
},

/*
 * Get the http URL
 */
getHTTPEndpoint : function(){
    return 'http://' + this.URL + ":" + this.PORT;
}

module.exports = config;

```

user.js : - Archivo de modelo donde se define el esquema

```

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

//Schema for User
var UserSchema = new Schema({
  name: {
    type: String,
    // required: true
  },
  email: {
    type: String
  },
  password: {
    type: String,
    //required: true
  },
  dob: {
    type: Date,
    //required: true
  },
  gender: {
    type: String, // Male/Female
    // required: true
  }
});

//Define the model for User
var User;
if(mongoose.models.User)
  User = mongoose.model('User');
else
  User = mongoose.model('User', UserSchema);

```

```
//Export the User Model
module.exports = User;
```

UserController : este archivo contiene la función para el registro de usuario

```
var User = require('../models/user');
var crypto = require('crypto');

//Controller for User
var UserController = {

  //Create a User
  create: function(req, res){
    var repassword = req.body.repassword;
    var password = req.body.password;
    var userEmail = req.body.email;

    //Check if the email address already exists
    User.find({"email": userEmail}, function(err, usr){
      if(usr.length > 0){
        //Email Exists

        res.json('Email already exists');
        return;
      }
      else
      {
        //New Email

        //Check for same passwords
        if(password != repassword){
          res.json('Passwords does not match');
          return;
        }

        //Generate Password hash based on sha1
        var shasum = crypto.createHash('sha1');
        shasum.update(req.body.password);
        var passwordHash = shasum.digest('hex');

        //Create User
        var user = new User();
        user.name = req.body.name;
        user.email = req.body.email;
        user.password = passwordHash;
        user.dob = Date.parse(req.body.dob) || "";
        user.gender = req.body.gender;

        //Validate the User
        user.validate(function(err){
          if(err){
            res.json(err);
            return;
          }else{
            //Finally save the User
            user.save(function(err){
              if(err)
              {
                res.json(err);
              }
            });
          }
        });
      }
    });
  }
};
```

```

        return;
    }

    //Remove Password before sending User details
    user.password = undefined;
    res.json(user);
    return;
});

    }
});
}
});
}

module.exports = UserController;

```

userRoutes.js : - Esta es la ruta para userController

```

var express = require('express');
var UserController = require('../controllers/userController');

//Routes for User
var UserRoutes = function(app)
{
    var router = express.Router();

    router.route('/users')
        .post(UserController.create);

    return router;
}

module.exports = UserRoutes;

```

El ejemplo anterior puede parecer demasiado grande, pero si un principiante en node.js con una pequeña combinación de conocimiento expreso intenta pasar por esto, lo encontrará fácil y realmente útil.

Lea Buen estilo de codificación en línea: <https://riptutorial.com/es/node-js/topic/6489/buen-estilo-de-codificacion>

Capítulo 14: Carga automática en los cambios

Examples

Carga automática de cambios en el código fuente usando nodemon

El paquete nodemon permite recargar automáticamente su programa cuando modifica cualquier archivo en el código fuente.

Instalando nodemon globalmente

```
npm install -g nodemon (o npm i -g nodemon)
```

Instalando nodemon localmente

En caso de que no quieras instalarlo globalmente.

```
npm install --save-dev nodemon (o npm i -D nodemon)
```

Usando nodemon

Ejecute su programa con `nodemon entry.js` (o `nodemon entry`)

Esto reemplaza el uso habitual de `node entry.js` (o `node entry`).

También puede agregar su inicio de nodemon como un script npm, lo que puede ser útil si desea proporcionar parámetros y no escribirlos cada vez.

Añadir **package.json**:

```
"scripts": {  
  "start": "nodemon entry.js -devmode -something 1"  
}
```

De esta manera solo puedes usar `npm start` desde tu consola.

Browsersync

Visión general

[Browsersync](#) es una herramienta que permite ver archivos en vivo y recargar el navegador. Está disponible como un [paquete NPM](#).

Instalación

Para instalar Browsersync, primero debes tener [Node.js](#) y NPM instalados. Para obtener más información, consulte la documentación de SO sobre [Instalación y ejecución de Node.js](#).

Una vez que su proyecto esté configurado, puede instalar Browsersync con el siguiente comando:

```
$ npm install browser-sync -D
```

Esto instalará Browsersync en el directorio local `node_modules` y lo guardará en sus dependencias de desarrollador.

Si prefiere instalarlo globalmente, use el indicador `-g` en lugar del indicador `-D`.

Usuarios de Windows

Si tiene problemas para instalar Browsersync en Windows, es posible que deba instalar Visual Studio para poder acceder a las herramientas de compilación para instalar Browsersync. A continuación, deberá especificar la versión de Visual Studio que está utilizando como:

```
$ npm install browser-sync --msvs_version=2013 -D
```

Este comando especifica la versión 2013 de Visual Studio.

Uso básico

Para volver a cargar automáticamente su sitio cada vez que cambie un archivo JavaScript en su proyecto, use el siguiente comando:

```
$ browser-sync start --proxy "myproject.dev" --files "**/*.js"
```

Reemplace `myproject.dev` con la dirección web que está utilizando para acceder a su proyecto. Browsersync generará una dirección alternativa que se puede usar para acceder a su sitio a través del proxy.

Uso avanzado

Además de la interfaz de línea de comandos que se describió anteriormente, Browsersync también se puede usar con [Grunt.js](#) y [Gulp.js](#).

Grunt.js

El uso con Grunt.js requiere un complemento que se pueda instalar así:

```
$ npm install grunt-browser-sync -D
```

Luego agregará esta línea a su `gruntfile.js` :

```
grunt.loadNpmTasks('grunt-browser-sync');
```

Gulp.js

Browsersync funciona como un módulo [CommonJS](#) , por lo que no hay necesidad de un complemento Gulp.js. Simplemente requiere el módulo así:

```
var browserSync = require('browser-sync').create();
```

Ahora puede utilizar la [API de Browsersync](#) para configurarlo según sus necesidades.

API

La API de Browsersync se puede encontrar aquí: <https://browsersync.io/docs/api>

Lea [Carga automática en los cambios en línea](#): <https://riptutorial.com/es/node-js/topic/1743/carga-automatica-en-los-cambios>

Capítulo 15: Casos de uso de Node.js

Examples

Servidor HTTP

```
const http = require('http');

console.log('Starting server...');
var config = {
  port: 80,
  contentType: 'application/json; charset=utf-8'
};
// JSON-API server on port 80

var server = http.createServer();
server.listen(config.port);
server.on('error', (err) => {
  if (err.code === 'EADDRINUSE') console.error('Port ' + config.port + ' is already in use');
  else console.error(err.message);
});
server.on('request', (request, res) => {
  var remoteAddress = request.headers['x-forwarded-for'] ||
  request.connection.remoteAddress; // Client address
  console.log(remoteAddress + ' ' + request.method + ' ' + request.url);

  var out = {};
  // Here you can change output according to `request.url`
  out.test = request.url;
  res.writeHead(200, {
    'Content-Type': config.contentType
  });
  res.end(JSON.stringify(out));
});
server.on('listening', () => {
  c.info('Server is available: http://localhost:' + config.port);
});
```

Consola con el símbolo del sistema

```
const process = require('process');
const rl = require('readline').createInterface(process.stdin, process.stdout);

rl.pause();
console.log('Something long is happening here...');

var cliConfig = {
  promptPrefix: ' > '
}

/*
  Commands recognition
  BEGIN
*/
var commands = {
```

```

eval: function(arg) { // Try typing in console: eval 2 * 10 ^ 3 + 2 ^ 4
    arg = arg.join(' ');
    try { console.log(eval(arg)); }
    catch (e) { console.log(e); }
},
exit: function(arg) {
    process.exit();
}
};
rl.on('line', (str) => {
    rl.pause();
    var arg = str.trim().match(/[^\s]+|"(?:[^\s\\]|\\.)+"/g); // Applying regular expression
    for removing all spaces except for what between double quotes:
    http://stackoverflow.com/a/14540319/2396907
    if (arg) {
        for (let n in arg) {
            arg[n] = arg[n].replace(/^\s|\s$/g, '');
        }
        var commandName = arg[0];
        var command = commands[commandName];
        if (command) {
            arg.shift();
            command(arg);
        }
        else console.log('Command "' + commandName + '" doesn\'t exist');
    }
    rl.prompt();
});
/*
    END OF
    Commands recognition
*/

rl.setPrompt(cliConfig.promptPrefix);
rl.prompt();

```

Lea Casos de uso de Node.js en línea: <https://riptutorial.com/es/node-js/topic/7703/casos-de-uso-de-node-js>

Capítulo 16: Cierre agraciado

Examples

Cierre agraciado - SIGTERM

Al usar **server.close ()** y **process.exit ()** , podemos detectar la excepción del servidor y hacer un cierre correcto.

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

var server = http.createServer(function (req, res) {
  setTimeout(function () { //simulate a long request
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
  }, 4000);
}).listen(9090, function (err) {
  console.log('listening http://localhost:9090/');
  console.log('pid is ' + process.pid);
});

process.on('SIGTERM', function () {
  server.close(function () {
    process.exit(0);
  });
});
```

Lea Cierre agraciado en línea: <https://riptutorial.com/es/node-js/topic/5996/cierre-agraciado>

Capítulo 17: CLI

Sintaxis

- `nodo [opciones] [opciones v8] [script.js | -e "script"] [argumentos]`

Examples

Opciones de línea de comando

```
-v, --version
```

Añadido en: v0.1.3 Imprimir versión del nodo.

```
-h, --help
```

Añadido en: v0.1.3 Imprimir opciones de línea de comando del nodo. El resultado de esta opción es menos detallado que este documento.

```
-e, --eval "script"
```

Añadido en: v0.5.2 Evalúe el siguiente argumento como JavaScript. Los módulos que están predefinidos en el REPL también se pueden usar en el script.

```
-p, --print "script"
```

Añadido en: v0.6.4 Idéntico a -e pero imprime el resultado.

```
-c, --check
```

Añadido en: v5.0.0 Sintaxis, compruebe el script sin ejecutar.

```
-i, --interactive
```

Añadido en: v0.7.7 Abre el REPL incluso si la entrada estándar no parece ser un terminal.

```
-r, --require module
```

Añadido en: v1.6.0 Precargue el módulo especificado al inicio.

Los seguimientos requieren las reglas de resolución de módulos de (). módulo puede ser una ruta a un archivo o un nombre de módulo de nodo.

```
--no-deprecation
```

Agregado en: v0.8.0 Advertencias de desaprobación de silencio.

```
--trace-deprecation
```

Agregado en: v0.8.0 Imprimir seguimientos de pila para desaprobaciones.

```
--throw-deprecation
```

Agregado en: v0.11.14 Tirar errores por desaprobaciones.

```
--no-warnings
```

Añadido en: v6.0.0 Silencie todas las advertencias de proceso (incluidas las depreciaciones).

```
--trace-warnings
```

Agregado en: v6.0.0 Imprimir seguimientos de la pila para advertencias de proceso (incluidas las desaprobaciones).

```
--trace-sync-io
```

Agregado en: v2.1.0 Imprime un seguimiento de pila cada vez que se detecta una E / S sincrónica después del primer giro del bucle de eventos.

```
--zero-fill-buffers
```

Añadido en: v6.0.0 Rellena automáticamente en cero todas las instancias de Buffer y SlowBuffer recién asignadas.

```
--preserve-symlinks
```

Añadido en: v6.3.0 Indica al cargador de módulos que mantenga los enlaces simbólicos al resolver y almacenar en caché los módulos.

De forma predeterminada, cuando Node.js carga un módulo desde una ruta que está simbólicamente vinculada a una ubicación diferente en el disco, Node.js eliminará la referencia al enlace y usará la "ruta real" real del disco como módulo. y como ruta raíz para ubicar otros módulos de dependencia. En la mayoría de los casos, este comportamiento predeterminado es aceptable. Sin embargo, cuando se usan dependencias de iguales vinculadas simbólicamente, como se ilustra en el siguiente ejemplo, el comportamiento predeterminado hace que se genere una excepción si el módulo A intenta requerir el módulo B como una dependencia de iguales:

```
{appDir}
├─ app
│   ├─ index.js
│   └─ node_modules
│       ├─ moduleA -> {appDir}/moduleA
│       └─ moduleB
```

```
├── index.js
├── package.json
└── moduleA
    ├── index.js
    └── package.json
```

El indicador de línea de comando `--preserve-symlinks` le indica a Node.js que use la ruta del enlace simbólico para los módulos en lugar de la ruta real, lo que permite que se encuentren dependencias entre iguales simbólicamente vinculadas.

Tenga en cuenta, sin embargo, que el uso de `--preserve-symlinks` puede tener otros efectos secundarios. Específicamente, los módulos nativos vinculados simbólicamente pueden no cargarse si están vinculados desde más de una ubicación en el árbol de dependencias (Node.js los vería como dos módulos separados e intentaría cargar el módulo varias veces, lo que provocaría una excepción)).

```
--track-heap-objects
```

Añadido en: v2.4.0 Rastrear las asignaciones de objetos del montón para las instantáneas del montón.

```
--prof-process
```

Añadido en: v6.0.0 Procesar la salida del generador de perfiles de v8 generada mediante la opción `v8 --prof`.

```
--v8-options
```

Añadido en: v0.1.3 Imprimir v8 opciones de línea de comando.

Nota: las opciones de v8 permiten que las palabras estén separadas por guiones (-) o guiones bajos (_).

Por ejemplo, `--stack-trace-limit` es equivalente a `--stack_trace_limit`.

```
--tls-cipher-list=list
```

Añadido en: v4.0.0 Especifique una lista de cifrado TLS predeterminada alternativa. (Requiere que Node.js sea creado con soporte criptográfico. (Predeterminado))

```
--enable-fips
```

Añadido en: v6.0.0 Habilitar criptografía compatible con FIPS en el inicio. (Requiere que Node.js sea construido con `./configure --openssl-fips`)

```
--force-fips
```

Añadido en: v6.0.0 Forzar el cifrado compatible con FIPS en el inicio. (No se puede desactivar

desde el código del script). (Los mismos requisitos que --enable-fips)

```
--icu-data-dir=file
```

Añadido en: v0.11.15 Especifique la ruta de carga de datos de ICU. (anula NODE_ICU_DATA)

```
Environment Variables
```

```
NODE_DEBUG=module[,...]
```

Agregado en: v0.1.32 ';' - lista separada de módulos principales que deberían imprimir información de depuración.

```
NODE_PATH=path[:...]
```

Agregado en: v0.1.32 ':' - lista separada de directorios prefijados a la ruta de búsqueda del módulo.

Nota: en Windows, esta es una lista separada por ';'.

```
NODE_DISABLE_COLORS=1
```

Añadido en: v0.3.0 Cuando se establece en 1, los colores no se utilizarán en el REPL.

```
NODE_ICU_DATA=file
```

Añadido en: v0.11.15 Ruta de datos para datos de ICU (objeto internacional). Extenderá los datos vinculados cuando se compilen con soporte de icu pequeño.

```
NODE_REPL_HISTORY=file
```

Añadido en: v5.0.0 Ruta al archivo utilizado para almacenar el historial de REPL persistente. La ruta predeterminada es ~ / .node_repl_history, que se invalida con esta variable. Establecer el valor en una cadena vacía ("" o "") deshabilita el historial de REPL persistente.

Lea CLI en línea: <https://riptutorial.com/es/node-js/topic/6013/cli>

Capítulo 18: Código Node.js para STDIN y STDOUT sin usar ninguna biblioteca

Introducción

Este es un programa simple en node.js al cual toma información del usuario y la imprime en la consola.

El objeto de **proceso** es un objeto global que proporciona información y control sobre el proceso Node.js actual. Como global, siempre está disponible para las aplicaciones Node.js sin usar `require ()`.

Examples

Programa

La propiedad **process.stdin** devuelve una secuencia legible equivalente o asociada a la entrada estándar.

La propiedad **process.stdout** devuelve una secuencia de escritura equivalente o asociada a stdout.

```
process.stdin.resume()
console.log('Enter the data to be displayed ');
process.stdin.on('data', function(data) { process.stdout.write(data) })
```

Lea Código Node.js para STDIN y STDOUT sin usar ninguna biblioteca en línea:

<https://riptutorial.com/es/node-js/topic/8961/codigo-node-js-para-stdin-y-stdout-sin-usar-ninguna-biblioteca>

Capítulo 19: Comenzando con el perfilado de nodos

Introducción

El objetivo de esta publicación es comenzar a perfilar la aplicación nodejs y cómo dar sentido a estos resultados para capturar un error o una pérdida de memoria. Una aplicación de ejecución nodejs no es más que un proceso del motor v8 que en muchos términos es similar a un sitio web que se ejecuta en un navegador y básicamente podemos capturar todas las métricas relacionadas con el proceso de un sitio web para una aplicación de nodo.

La herramienta de mi preferencia es devtools de cromo o inspector de cromo junto con el inspector de nodos.

Observaciones

El inspector de nodos falla al adjuntar al proceso de debug del nodo a veces, en cuyo caso no podrá obtener el punto de interrupción de depuración en devtools. Pruebe a actualizar la pestaña de devtools varias veces y espere unos segundos para ver si está en modo de depuración.

Si no es así, reinicie el inspector de nodos desde la línea de comando.

Examples

Perfilando una aplicación de nodo simple

Paso 1 : instale el paquete node-inspector usando npm globalmente en su máquina

```
$ npm install -g node-inspector
```

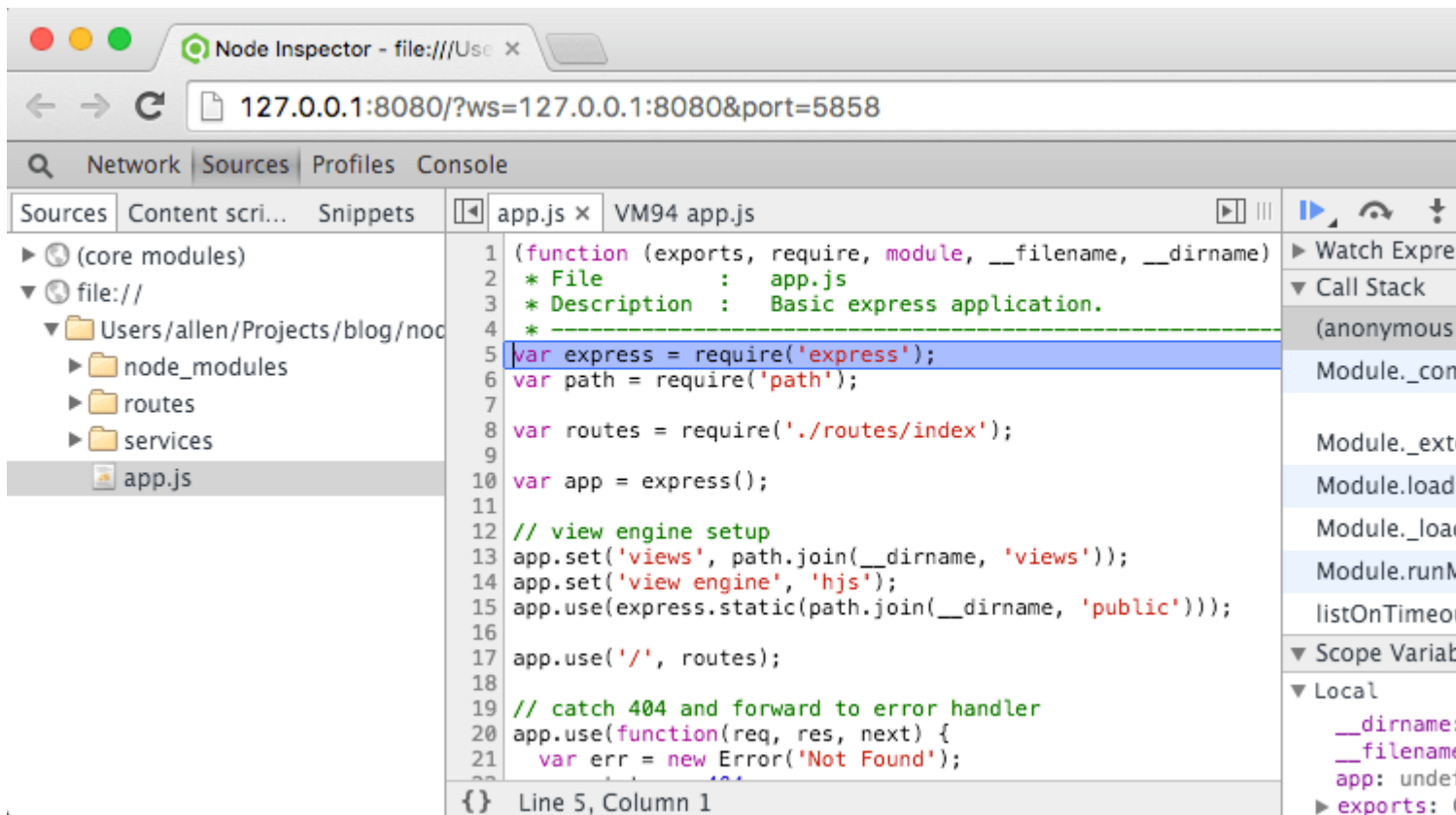
Paso 2 : Iniciar el servidor de inspector de nodos

```
$ node-inspector
```

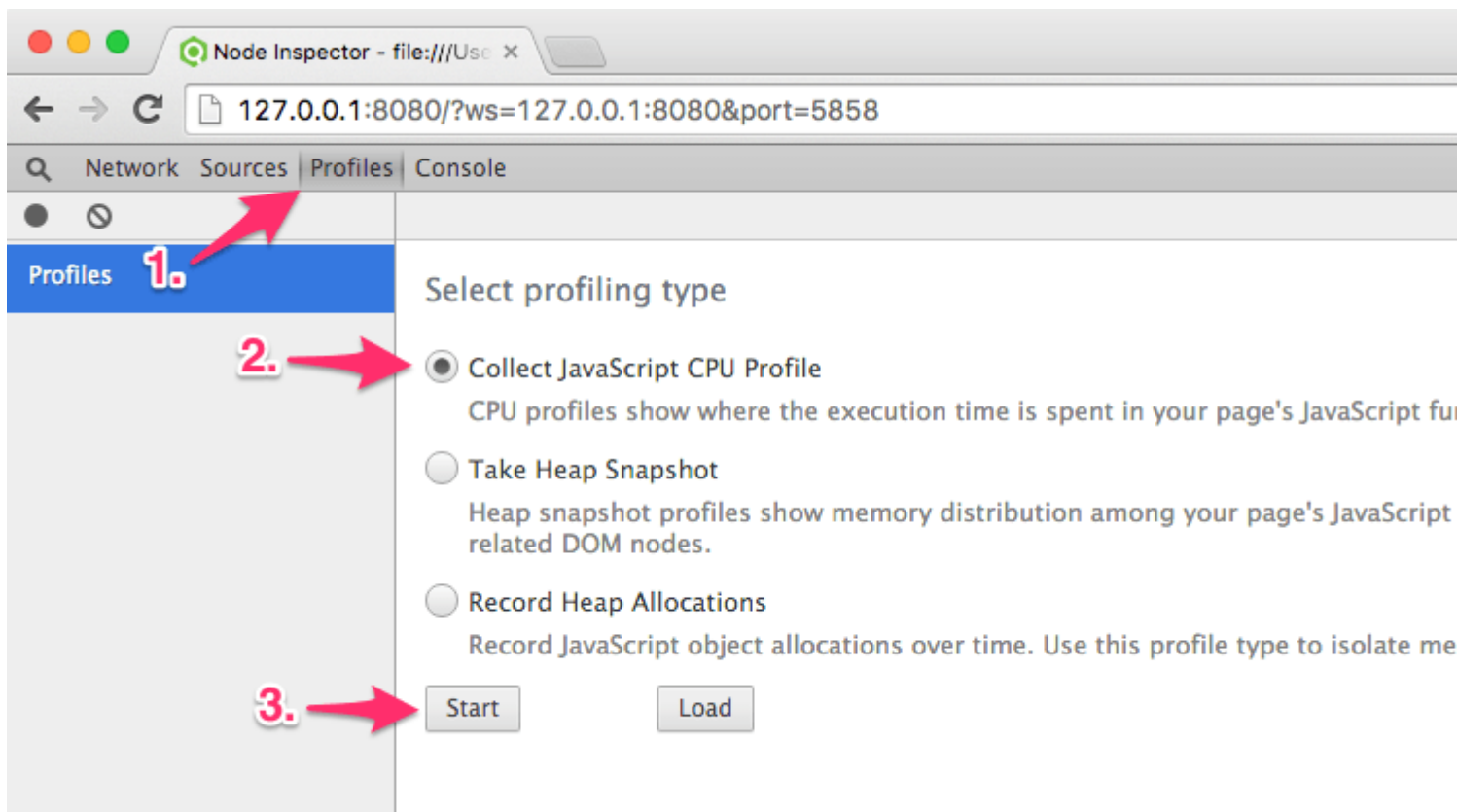
Paso 3 : Comience a depurar su aplicación de nodo

```
$ node --debug-brk your/short/node/script.js
```

Paso 4 : abre <http://127.0.0.1:8080/?port=5858> en el navegador Chrome. Y verá una interfaz de herramientas chrom-dev con el código fuente de su aplicación nodejs en el panel izquierdo. Y dado que hemos utilizado la opción de ruptura de depuración al depurar la aplicación, la ejecución del código se detendrá en la primera línea de código.



Paso 5 : Esta es la parte fácil en la que cambia a la pestaña de perfiles y comienza a perfilar la aplicación. En el caso de que desee obtener el perfil para un método o flujo en particular, asegúrese de que la ejecución del código sea un punto crítico justo antes de que se ejecute ese fragmento de código.



Paso 6 : Una vez que haya registrado su perfil de CPU, su volcado de pila / instantánea o la

asignación de pila, puede ver los resultados en la misma ventana o guardarlos en una unidad local para su posterior análisis o comparación con otros perfiles.

Puedes usar estos artículos para saber leer los perfiles:

- [Lectura de perfiles de CPU](#)
- [Chrome Profiler CPU y Heap Profiler](#)

Lea [Comenzando con el perfilado de nodos en línea](#): <https://riptutorial.com/es/node-js/topic/9347/comenzando-con-el-perfilado-de-nodos>

Capítulo 20: Cómo se cargan los módulos

Examples

Modo global

Si instaló Node usando el directorio predeterminado, mientras estaba en el modo global, NPM instala paquetes en `/usr/local/lib/node_modules`. Si escribe lo siguiente en el shell, NPM buscará, descargará e instalará la última versión del paquete llamado `sax` dentro del directorio

`/usr/local/lib/node_modules/express`:

```
$ npm install -g express
```

Asegúrese de tener suficientes derechos de acceso a la carpeta. Estos módulos estarán disponibles para todos los procesos de nodo que se ejecutarán en esa máquina

En modo de instalación local. Npm descargará e instalará módulos en las carpetas de trabajo actuales al crear una nueva carpeta llamada `node_modules` por ejemplo, si está en

`/home/user/apps/my_app` una nueva carpeta llamada `node_modules`

`/home/user/apps/my_app/node_modules` si aún no existen

Cargando módulos

Cuando hacemos referencia al módulo en el código, el nodo primero busca la carpeta `node_module` dentro de la carpeta referenciada en la declaración requerida. Si el nombre del módulo no es relativo y no es un módulo central, Node intentará encontrarlo dentro de la carpeta `node_modules` en la `node_modules` actual. directorio. Por ejemplo, si hace lo siguiente, Node intentará buscar el archivo `./node_modules/myModule.js`:

```
var myModule = require('myModule.js');
```

Si Node no encuentra el archivo, buscará dentro de la carpeta principal llamada `./node_modules/myModule.js`. Si vuelve a fallar, intentará con la carpeta principal y seguirá descendiendo hasta que llegue a la raíz o encuentre el módulo requerido.

También puede omitir la extensión `.js` si lo desea, en cuyo caso el nodo agregará la extensión `.js` y buscará el archivo.

Cargando un módulo de carpeta

Puede usar la ruta de una carpeta para cargar un módulo como este:

```
var myModule = require('./myModuleDir');
```

Si lo haces, Node buscará dentro de esa carpeta. Node supondrá que esta carpeta es un paquete e intentará buscar una definición de paquete. Esa definición de paquete debe ser un archivo llamado `package.json` . Si esa carpeta no contiene un archivo de definición de paquete llamado `package.json` , el punto de entrada del paquete asumirá el valor predeterminado de `index.js` , y Node buscará, en este caso, un archivo bajo la ruta `./myModuleDir/index.js`

El último recurso si el módulo no se encuentra en ninguna de las carpetas es la carpeta de instalación del módulo global.

Lea Cómo se cargan los módulos en línea: <https://riptutorial.com/es/node-js/topic/7738/como-se-cargan-los-modulos>

Capítulo 21: Comunicación cliente-servidor

Examples

/ w Express, jQuery y Jade

```
//'client.jade'

//a button is placed down; similar in HTML
button(type='button', id='send_by_button') Modify data

#modify Lorem ipsum Sender

//loading jQuery; it can be done from an online source as well
script(src='../js/jquery-2.2.0.min.js')

//AJAX request using jQuery
script
  $(function () {
    $('#send_by_button').click(function (e) {
      e.preventDefault();

      //test: the text within brackets should appear when clicking on said button
      //window.alert('You clicked on me. - jQuery');

      //a variable and a JSON initialized in the code
      var predeclared = "Katamori";
      var data = {
        Title: "Name_SenderTest",
        Nick: predeclared,
        FirstName: "Zoltan",
        Surname: "Schmidt"
      };

      //an AJAX request with given parameters
      $.ajax({
        type: 'POST',
        data: JSON.stringify(data),
        contentType: 'application/json',
        url: 'http://localhost:7776/domainetest',

        //on success, received data is used as 'data' function input
        success: function (data) {
          window.alert('Request sent; data received.');
```

```

        });
    });
});

// 'domaintest_route.js'

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

// an Express router listening to GET requests - in this case, it's empty, meaning that nothing
// is displayed when you reach 'localhost/domaintest'
router.get('/', function(req, res, next) {
});

// same for POST requests - notice, how the AJAX request above was defined as POST
router.post('/', function(req, res) {
    res.setHeader('Content-Type', 'application/json');

    // content generated here
    var some_json = {
        Title: "Test",
        Item: "Crate"
    };

    var result = JSON.stringify(some_json);

    // content got 'client.jade'
    var sent_data = req.body;
    sent_data.Nick = "ttony33";

    res.send(sent_data);
});

module.exports = router;

```

// basado en un gist usado personalmente:

<https://gist.github.com/Katamori/5c9850f02e4baf6e9896>

Lea Comunicación cliente-servidor en línea: <https://riptutorial.com/es/node-js/topic/6222/comunicacion-cliente-servidor>

Capítulo 22: Comunicación socket.io

Examples

"¡Hola Mundo!" Con mensajes de socket.

Instalar módulos de nodo

```
npm install express
npm install socket.io
```

Servidor Node.js

```
const express = require('express');
const app = express();
const server = app.listen(3000, console.log("Socket.io Hello World server started!"));
const io = require('socket.io')(server);

io.on('connection', (socket) => {
  //console.log("Client connected!");
  socket.on('message-from-client-to-server', (msg) => {
    console.log(msg);
  })
  socket.emit('message-from-server-to-client', 'Hello World!');
});
```

Cliente navegador

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Hello World with Socket.io</title>
  </head>
  <body>
    <script src="https://cdn.socket.io/socket.io-1.4.5.js"></script>
    <script>
      var socket = io("http://localhost:3000");
      socket.on("message-from-server-to-client", function(msg) {
        document.getElementById('message').innerHTML = msg;
      });
      socket.emit('message-from-client-to-server', 'Hello World!');
    </script>
    <p>Socket.io Hello World client started!</p>
    <p id="message"></p>
  </body>
</html>
```

Lea Comunicación socket.io en línea: <https://riptutorial.com/es/node-js/topic/4261/comunicacion-socket-io>

Capítulo 23: Conectarse a MongoDB

Introducción

MongoDB es un programa de base de datos orientado a documentos y multiplataforma gratuito y de código abierto. Clasificado como un programa de base de datos NoSQL, MongoDB usa documentos similares a JSON con esquemas.

Para más detalles, vaya a <https://www.mongodb.com/>

Sintaxis

- `MongoClient.connect('mongodb://127.0.0.1: 27017 / crud', function (err, db) { // do something here});`

Examples

Ejemplo simple para conectar mongoDB desde Node.JS

```
MongoClient.connect('mongodb://localhost:27017/myNewDB',function (err,db) {
  if(err)
    console.log("Unable to connect DB. Error: " + err)
  else
    console.log('Connected to DB');

  db.close();
});
```

myNewDB es el nombre de la base de datos, si no existe en la base de datos, se creará automáticamente con esta llamada.

Una forma sencilla de conectar mongoDB con núcleo Node.JS

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

//connection with mongoDB
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {
  //check the connection
  if(err){
    console.log("connection failed.");
  }else{
    console.log("successfully connected to mongoDB.");
  }
});
```

Lea Conectarse a MongoDB en línea: <https://riptutorial.com/es/node-js/topic/6280/conectarse-a-mongodb>

Capítulo 24: Conexión Mysql Pool

Examples

Usando un grupo de conexiones sin base de datos

Lograr la multitenidad en el servidor de bases de datos con múltiples bases de datos alojadas en él.

La multipropiedad es un requisito común de las aplicaciones empresariales en la actualidad y no se recomienda crear un grupo de conexiones para cada base de datos en el servidor de bases de datos. Entonces, lo que podemos hacer es crear un grupo de conexiones con el servidor de base de datos y luego cambiar entre las bases de datos alojadas en el servidor de base de datos a pedido.

Supongamos que nuestra aplicación tiene diferentes bases de datos para cada empresa alojada en el servidor de bases de datos. Nos conectaremos a la base de datos de la empresa respectiva cuando el usuario acceda a la aplicación. Aquí está el ejemplo de cómo hacer eso:

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
});

pool.getConnection(function(err, connection){
  if(err){
    return cb(err);
  }
  connection.changeUser({database : "firm1"});
  connection.query("SELECT * from history", function(err, data){
    connection.release();
    cb(err, data);
  });
});
```

Déjame desglosar el ejemplo:

Al definir la configuración del grupo, no le di el nombre de la base de datos, sino que solo le di un servidor de base de datos, es decir

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

por eso, cuando queremos usar la base de datos específica en el servidor de la base de datos,

pedimos la conexión para golpear la base de datos mediante:

```
connection.changeUser({database : "firm1"});
```

Puede consultar la documentación oficial [aquí](#).

Lea Conexión Mysql Pool en línea: <https://riptutorial.com/es/node-js/topic/6353/conexion-mysql-pool>

Capítulo 25: Cortar

Examples

Añadir nuevas extensiones para `require()`

Puede agregar nuevas extensiones a `require()` extendiendo `require.extensions`.

Para un ejemplo de **XML** :

```
// Add .xml for require()
require.extensions['.xml'] = (module, filename) => {
  const fs = require('fs')
  const xml2js = require('xml2js')

  module.exports = (callback) => {
    // Read required file.
    fs.readFile(filename, 'utf8', (err, data) => {
      if (err) {
        callback(err)
        return
      }
      // Parse it.
      xml2js.parseString(data, (err, result) => {
        callback(null, result)
      })
    })
  }
}
```

Si el contenido de `hello.xml` es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
  <bar>baz</bar>
  <qux />
</foo>
```

Puedes leerlo y analizarlo a través de `require()` :

```
require('./hello')((err, xml) {
  if (err)
    throw err;
  console.log(err);
})
```

Imprime `{ foo: { bar: [ 'baz' ], qux: [ '' ] } }`.

Lea Cortar en línea: <https://riptutorial.com/es/node-js/topic/6645/cortar>

Capítulo 26: Creación de una biblioteca Node.js que admita tanto las promesas como las devoluciones de llamada de error primero

Introducción

A muchas personas les gusta trabajar con promesas y / o sintaxis asíncronica / a la espera, pero al escribir un módulo también sería útil para algunos programadores admitir métodos clásicos de estilo de devolución de llamada. En lugar de crear dos módulos, o dos conjuntos de funciones, o hacer que el programador prometa su módulo, su módulo puede admitir ambos métodos de programación a la vez usando `asCallback ()` de Bluebird o bien `nodeify ()` de Q.

Examples

Módulo de ejemplo y programa correspondiente usando Bluebird

math.js

```
'use strict';

const Promise = require('bluebird');

module.exports = {

  // example of a callback-only method
  callbackSum: function(a, b, callback) {
    if (typeof a !== 'number')
      return callback(new Error('"a" must be a number'));
    if (typeof b !== 'number')
      return callback(new Error('"b" must be a number'));

    return callback(null, a + b);
  },

  // example of a promise-only method
  promiseSum: function(a, b) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error('"a" must be a number'));
      if (typeof b !== 'number')
        return reject(new Error('"b" must be a number'));
      resolve(a + b);
    });
  },

  // a method that can be used as a promise or with callbacks
  sum: function(a, b, callback) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error('"a" must be a number'));
    });
  }
};
```

```

        if (typeof b !== 'number')
            return reject(new Error('"b" must be a number'));
        resolve(a + b);
    }).asCallback(callback);
},

};

```

index.js

```

'use strict';

const math = require('./math');

// classic callbacks

math.callbackSum(1, 3, function(err, result) {
    if (err)
        console.log('Test 1: ' + err);
    else
        console.log('Test 1: the answer is ' + result);
});

math.callbackSum(1, 'd', function(err, result) {
    if (err)
        console.log('Test 2: ' + err);
    else
        console.log('Test 2: the answer is ' + result);
});

// promises

math.promiseSum(2, 5)
    .then(function(result) {
        console.log('Test 3: the answer is ' + result);
    })
    .catch(function(err) {
        console.log('Test 3: ' + err);
    });

math.promiseSum(1)
    .then(function(result) {
        console.log('Test 4: the answer is ' + result);
    })
    .catch(function(err) {
        console.log('Test 4: ' + err);
    });

// promise/callback method used like a promise

math.sum(8, 2)
    .then(function(result) {
        console.log('Test 5: the answer is ' + result);
    })
    .catch(function(err) {
        console.log('Test 5: ' + err);
    });

```

```
// promise/callback method used with callbacks

math.sum(7, 11, function(err, result) {
  if (err)
    console.log('Test 6: ' + err);
  else
    console.log('Test 6: the answer is ' + result);
});

// promise/callback method used like a promise with async/await syntax

(async () => {

  try {
    let x = await math.sum(6, 3);
    console.log('Test 7a: ' + x);

    let y = await math.sum(4, 's');
    console.log('Test 7b: ' + y);

  } catch(err) {
    console.log(err.message);
  }

})();
```

Lea Creación de una biblioteca Node.js que admita tanto las promesas como las devoluciones de llamada de error primero en línea: <https://riptutorial.com/es/node-js/topic/9874/creacion-de-una-biblioteca-node-js-que-admita-tanto-las-promesas-como-las-devoluciones-de-llamada-de-error-primero>

Capítulo 27: Creando API's con Node.js

Examples

OBTENER API utilizando Express

Node.js apis se puede construir fácilmente en el marco web Express .

El siguiente ejemplo crea una api GET simple para enumerar a todos los usuarios.

Ejemplo

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

var users =[
  {
    id: 1,
    name: "John Doe",
    age : 23,
    email: "john@doe.com"
  }
];

// GET /api/users
app.get('/api/users', function(req, res){
  return res.json(users);    //return response as JSON
});

app.listen('3000', function(){
  console.log('Server listening on port 3000');
});
```

POST API utilizando Express

El siguiente ejemplo crea la API POST usando Express . Este ejemplo es similar al ejemplo GET excepto el uso de body-parser de body-parser que analiza los datos de la publicación y los agrega a req.body .

Ejemplo

```
var express = require('express');
var app = express();
// for parsing the body in POST request
var bodyParser = require('body-parser');

var users =[
  {
    id: 1,
    name: "John Doe",
    age : 23,
    email: "john@doe.com"
  }
];

app.use(bodyParser.urlencoded({ extended: false }));
```

```
app.use(bodyParser.json());

// GET /api/users
app.get('/api/users', function(req, res){
  return res.json(users);
});

/* POST /api/users
  {
    "user": {
      "id": 3,
      "name": "Test User",
      "age" : 20,
      "email": "test@test.com"
    }
  }
*/
app.post('/api/users', function (req, res) {
  var user = req.body.user;
  users.push(user);

  return res.send('User has been added successfully');
});

app.listen('3000', function(){
  console.log('Server listening on port 3000');
});
```

Lea Creando API's con Node.js en línea: <https://riptutorial.com/es/node-js/topic/5991/creando-api-s-con-node-js>

Capítulo 28: csv parser en el nodo js

Introducción

La lectura de datos desde un csv se puede manejar de muchas maneras. Una solución es leer el archivo `csv` en una matriz. A partir de ahí puedes trabajar en la matriz.

Examples

Usando FS para leer en un CSV

fs es la [API del sistema de archivos](#) en el nodo. Podemos usar el método `readFile` en nuestra variable `fs`, pasarle un archivo `data.csv`, formato y función que lea y divida el `csv` para su posterior procesamiento.

Esto supone que tiene un archivo llamado `data.csv` en la misma carpeta.

```
'use strict'

const fs = require('fs');

fs.readFile('data.csv', 'utf8', function (err, data) {
  var dataArray = data.split(/\r?\n/);
  console.log(dataArray);
});
```

Ahora puede usar la matriz como cualquier otra para trabajar en ella.

Lea csv parser en el nodo js en línea: <https://riptutorial.com/es/node-js/topic/9162/csv-parser-en-el-nodo-js>

Capítulo 29: Depuración remota en Node.JS

Examples

Configuración de ejecución NodeJS

Para configurar la depuración remota del nodo, simplemente ejecute el proceso del nodo con el indicador `--debug` . Puede agregar un puerto en el que el depurador debe ejecutarse utilizando `--debug=<port>` .

Cuando su proceso de nodo se inicie debería ver el mensaje

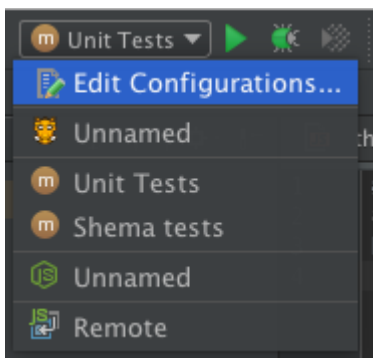
```
Debugger listening on port <port>
```

Lo que te dirá que todo está bien para ir.

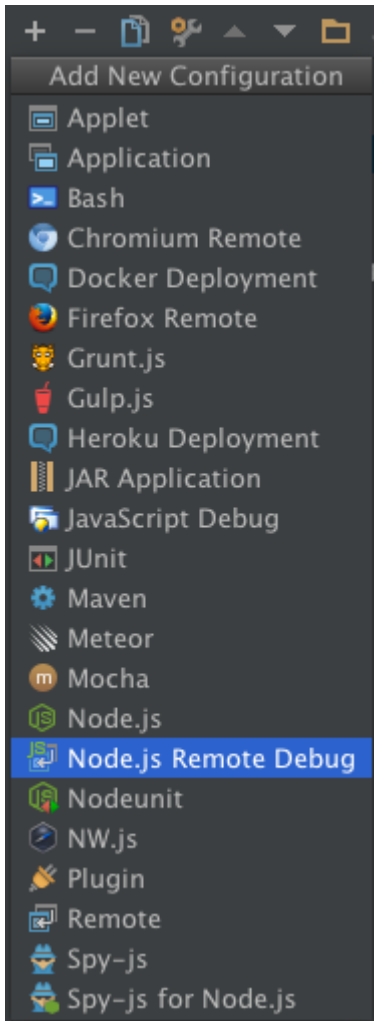
Luego, configura el destino de depuración remota en su IDE específico.

Configuración de IntelliJ / Webstorm

1. Asegúrese de que el complemento NodeJS esté habilitado
2. Seleccione sus configuraciones de ejecución (pantalla)



3. Seleccione **+ > Depuración remota Node.js**



4. Asegúrese de ingresar el puerto seleccionado arriba así como el host correcto

Name: ☐ Share ☒ Single instance only

Host:

Port:

Una vez que estén configurados, simplemente ejecute el destino de depuración como lo haría normalmente y se detendrá en sus puntos de interrupción.

Utilice el proxy para la depuración a través del puerto en Linux

Si inicia su aplicación en Linux, use el proxy para la depuración a través del puerto, por ejemplo:

```
socat TCP-LISTEN:9958,fork TCP:127.0.0.1:5858 &
```

Utilice el puerto 9958 para la depuración remota entonces.

Lea Depuración remota en Node.JS en línea: <https://riptutorial.com/es/node-js/topic/6335/depuracion-remota-en-node-js>

Capítulo 30: Depurando la aplicación Node.js

Examples

Core node.js depurador e inspector de nodos

Usando el depurador de núcleo

Node.js proporciona una compilación en una utilidad de depuración no gráfica. Para iniciar la compilación en el depurador, inicie la aplicación con este comando:

```
node debug filename.js
```

Considere la siguiente aplicación Node.js simple contenida en el `debugDemo.js`

```
'use strict';

function addTwoNumber(a, b){
  // function returns the sum of the two numbers
  debugger
  return a + b;
}

var result = addTwoNumber(5, 9);
console.log(result);
```

El `debugger` palabras clave detendrá al depurador en ese punto del código.

Referencia de comando

1. Paso a paso

```
cont, c - Continue execution
next, n - Step next
step, s - Step in
out, o - Step out
```

2. Puntos de interrupción

```
setBreakpoint(), sb() - Set breakpoint on current line
setBreakpoint(line), sb(line) - Set breakpoint on specific line
```

Para depurar el código anterior ejecute el siguiente comando

```
node debug debugDemo.js
```

Una vez que se ejecutan los comandos anteriores, verá la siguiente salida. Para salir de la interfaz del depurador, escriba `process.exit()`

```
ankuranand:~/workspace/nodejs/nodejsDebugging $ node debug debugDemo.js
< Debugger listening on port 5858
debug> . ok
break in debugDemo.js:3
  1 // A Demo Code Showing the basic capabilities of the nodejs  debugging module
  2
> 3 'use strict';
  4
  5 function addTwoNumber(a, b){
debug> n
break in debugDemo.js:11
  9 }
 10
>11 let result = addTwoNumber(5, 9);
 12 console.log(result);
 13
debug> c
break in debugDemo.js:7
  5 function addTwoNumber(a, b){
  6 // function returns the sum of the two numbers
> 7 debugger
  8   return a + b;
  9 }
debug> c
< 14
debug> process.exit()
ankuranand:~/workspace/nodejs/nodejsDebugging $
```

Use el comando `watch(expression)` para agregar la variable o expresión cuyo valor desea ver y `restart` para reiniciar la aplicación y la depuración.

Use `repl` para ingresar el código de manera interactiva. El modo de respuesta tiene el mismo contexto que la línea que está depurando. Esto le permite examinar el contenido de las variables y probar las líneas de código. Presione `Ctrl+C` para dejar la respuesta de depuración.

Usando el inspector de nodos incorporado

v6.3.0

Puede ejecutar el inspector v8 [incorporado en el nodo](#)! El complemento [inspector de nodos](#) ya no es necesario.

Simplemente pase el indicador de inspector y se le proporcionará una URL para el inspector

```
node --inspect server.js
```

Usando inspector de nodos

Instale el inspector de nodo:

```
npm install -g node-inspector
```

Ejecute su aplicación con el comando node-debug:

```
node-debug filename.js
```

Después de eso, pulsa en Chrome:

```
http://localhost:8080/debug?port=5858
```

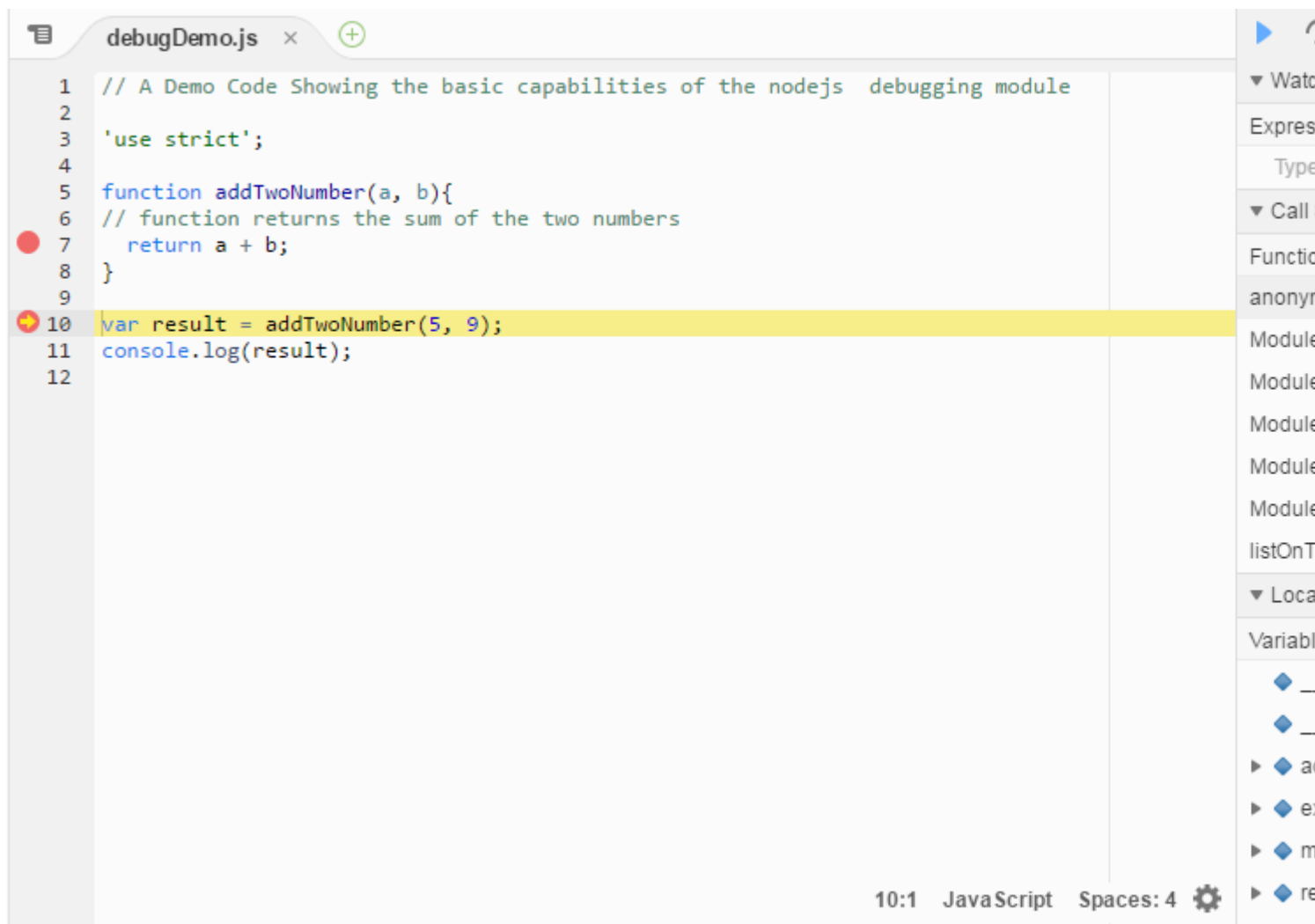
A veces, el puerto 8080 puede no estar disponible en su computadora. Puede obtener el siguiente error:

No se puede iniciar el servidor en 0.0.0.0:8080. Error: escuchar EACCES.

En este caso, inicie el inspector de nodos en un puerto diferente utilizando el siguiente comando.

```
$node-inspector --web-port=6500
```

Verás algo como esto:



Lea Depurando la aplicación Node.js en línea: <https://riptutorial.com/es/node-js/topic/5900/depurando-la-aplicacion-node-js>

Capítulo 31: Desafíos de rendimiento

Examples

Procesando consultas de larga ejecución con Nodo

Dado que Node es de un solo hilo, hay una solución alternativa si se trata de cálculos de larga ejecución.

Nota: este es el ejemplo "listo para ejecutar". Simplemente, no olvide obtener jQuery e instalar los módulos necesarios.

Lógica principal de este ejemplo:

1. Cliente envía solicitud al servidor.
2. El servidor inicia la rutina en una instancia de nodo separada y envía una respuesta inmediata con el ID de tarea relacionado.
3. El cliente continuamente envía cheques a un servidor para actualizaciones de estado de la ID de tarea dada.

Estructura del proyecto:

```
project
|   package.json
|   index.html
|
|---js
|   main.js
|   jquery-1.12.0.min.js
|
|---srv
|   app.js
|   models
|   |   task.js
|   |---tasks
|       data-processor.js
```

app.js:

```
var express      = require('express');
var app          = express();
var http         = require('http').Server(app);
var mongoose     = require('mongoose');
var bodyParser   = require('body-parser');

var childProcess= require('child_process');

var Task         = require('./models/task');

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

```

app.use(express.static(__dirname + '/../'));

app.get('/', function(request, response){
    response.render('index.html');
});

//route for the request itself
app.post('/long-running-request', function(request, response){
    //create new task item for status tracking
    var t = new Task({ status: 'Starting ...' });

    t.save(function(err, task){
        //create new instance of node for running separate task in another thread
        taskProcessor = childProcess.fork('./srv/tasks/data-processor.js');

        //process the messages coming from the task processor
        taskProcessor.on('message', function(msg){
            task.status = msg.status;
            task.save();
        }).bind(this));

        //remove previously opened node instance when we finished
        taskProcessor.on('close', function(msg){
            this.kill();
        });

        //send some params to our separate task
        var params = {
            message: 'Hello from main thread'
        };

        taskProcessor.send(params);
        response.status(200).json(task);
    });
});

//route to check is the request is finished the calculations
app.post('/is-ready', function(request, response){
    Task
        .findById(request.body.id)
        .exec(function(err, task){
            response.status(200).json(task);
        });
});

mongoose.connect('mongodb://localhost/test');
http.listen('1234');

```

task.js:

```

var mongoose = require('mongoose');

var taskSchema = mongoose.Schema({
    status: {
        type: String
    }
});

mongoose.model('Task', taskSchema);

```

```
module.exports = mongoose.model('Task');
```

data-processor.js:

```
process.on('message', function(msg){
  init = function(){
    processData(msg.message);
  }.bind(this)();

  function processData(message){
    //send status update to the main app
    process.send({ status: 'We have started processing your data.' });

    //long calculations ..
    setTimeout(function(){
      process.send({ status: 'Done!' });

      //notify node, that we are done with this task
      process.disconnect();
    }, 5000);
  }
});

process.on('uncaughtException', function(err){
  console.log("Error happened: " + err.message + "\n" + err.stack + ".\n");
  console.log("Gracefully finish the routine.");
});
```

index.html:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="../js/jquery-1.12.0.min.js"></script>
    <script src="../js/main.js"></script>
  </head>
  <body>
    <p>Example of processing long-running node requests.</p>
    <button id="go" type="button">Run</button>

    <br />

    <p>Log:</p>
    <textarea id="log" rows="20" cols="50"></textarea>
  </body>
</html>
```

main.js:

```
$(document).on('ready', function(){

  $('#go').on('click', function(e){
    //clear log
    $('#log').val('');

    $.post("/long-running-request", {some_params: 'params' })
      .done(function(task){
```

```

    $("#log").val( $("#log").val() + '\n' + task.status);

    //function for tracking the status of the task
    function updateStatus(){
        $.post("/is-ready", {id: task._id })
        .done(function(response){
            $("#log").val( $("#log").val() + '\n' + response.status);

            if(response.status !== 'Done!'){
                checkTaskTimeout = setTimeout(updateStatus, 500);
            }
        });
    }

    //start checking the task
    var checkTaskTimeout = setTimeout(updateStatus, 100);
});
});
});

```

paquete.json:

```

{
  "name": "nodeProcessor",
  "dependencies": {
    "body-parser": "^1.15.2",
    "express": "^4.14.0",
    "html": "0.0.10",
    "mongoose": "^4.5.5"
  }
}

```

Descargo de responsabilidad: este ejemplo pretende darle una idea básica. Para usarlo en el entorno de producción, necesita mejoras.

Lea Desafíos de rendimiento en línea: <https://riptutorial.com/es/node-js/topic/6325/desafios-de-rendimiento>

Capítulo 32: Desinstalar Node.js

Examples

Desinstale completamente Node.js en Mac OSX

En la Terminal de su sistema operativo Mac, ingrese los siguientes 2 comandos:

```
lsbom -f -l -s -pf /var/db/receipts/org.nodejs.pkg.bom | while read f; do sudo rm /usr/local/${f}; done  
  
sudo rm -rf /usr/local/lib/node /usr/local/lib/node_modules /var/db/receipts/org.nodejs.*
```

Desinstalar Node.js en Windows

Para desinstalar Node.js en Windows, use Agregar o quitar programas como este:

1. Abre `Add or Remove Programs` del menú de inicio.
2. Buscar `Node.js`

Windows 10:

3. Haga clic en Node.js.
4. Haga clic en Desinstalar.
5. Haga clic en el nuevo botón Desinstalar.

Windows 7-8.1:

3. Haga clic en el botón Desinstalar debajo de Node.js.

Lea Desinstalar Node.js en línea: <https://riptutorial.com/es/node-js/topic/2821/desinstalar-node-js>

Capítulo 33: Despliegue de aplicaciones Node.js en producción

Examples

Configurando `NODE_ENV = "producción"`

Las implementaciones de producción variarán de muchas maneras, pero una convención estándar cuando se implementa en producción es definir una variable de entorno llamada `NODE_ENV` y establecer su valor en *"producción"*.

Banderas de tiempo de ejecución

Cualquier código que se ejecute en su aplicación (incluidos los módulos externos) puede verificar el valor de `NODE_ENV`:

```
if(process.env.NODE_ENV === 'production') {  
  // We are running in production mode  
} else {  
  // We are running in development mode  
}
```

Dependencias

Cuando la variable de entorno `NODE_ENV` se establece en *'producción'*, todas las `devDependencies` en su archivo `package.json` se ignorarán completamente cuando se ejecute `npm install`. También puede imponer esto con un indicador de `--production`:

```
npm install --production
```

Para configurar `NODE_ENV` puedes usar cualquiera de estos métodos

Método 1: configurar `NODE_ENV` para todas las aplicaciones de nodo

Windows:

```
set NODE_ENV=production
```

Linux u otro sistema basado en Unix:

```
export NODE_ENV=production
```

Esto establece `NODE_ENV` para la sesión de bash actual, por lo tanto, cualquier aplicación iniciada

después de esta declaración tendrá `NODE_ENV` establecido en `production`.

Método 2: establece `NODE_ENV` para la aplicación actual

```
NODE_ENV=production node app.js
```

Esto establecerá `NODE_ENV` para la aplicación actual. Esto ayuda cuando queremos probar nuestras aplicaciones en diferentes entornos.

Método 3: crear `.env` archivo `.env` y usarlo

Esto utiliza la idea explicada [aquí](#). Consulte esta publicación para una explicación más detallada.

Básicamente, creas `.env` archivo `.env` y ejecutas algunos scripts de bash para establecerlos en el entorno.

Para evitar escribir un script bash, el paquete `env-cmd` se puede usar para cargar las variables de entorno definidas en el archivo `.env`.

```
env-cmd .env node app.js
```

Método 4: usar el paquete de `cross-env`

Este [paquete](#) permite que las variables de entorno se configuren de una manera para cada plataforma.

Después de instalarlo con npm, puede agregarlo a su script de implementación en `package.json` siguiente manera:

```
"build:deploy": "cross-env NODE_ENV=production webpack"
```

Administrar la aplicación con el administrador de procesos

Es una buena práctica ejecutar aplicaciones de NodeJS controladas por los administradores de procesos. El administrador de procesos ayuda a mantener viva la aplicación para siempre, reinicia en caso de falla, recarga sin tiempo de inactividad y simplifica la administración. Los más poderosos de ellos (como [PM2](#)) tienen un equilibrador de carga incorporado. PM2 también le permite administrar el registro de aplicaciones, la supervisión y la agrupación en clústeres.

Gestor de procesos PM2

Instalación de PM2:

```
npm install pm2 -g
```

El proceso se puede iniciar en modo de clúster que incluye un equilibrador de carga integrado para distribuir la carga entre procesos:

```
pm2 start app.js -i 0 --name "api" (-i es para especificar el número de procesos que se
```

generarán. Si es 0, el número de proceso se basará en el recuento de núcleos de CPU)

Si bien tiene múltiples usuarios en producción, es necesario tener un solo punto para PM2. Por lo tanto, el comando pm2 debe tener el prefijo de una ubicación (para la configuración de PM2), de lo contrario, generará un nuevo proceso de pm2 para cada usuario con la configuración en el directorio de inicio correspondiente. Y será inconsistente.

Uso: PM2_HOME=/etc/.pm2 pm2 start app.js

Despliegue utilizando PM2

PM2 es un administrador de procesos de producción para aplicaciones `Node.js`, que le permite mantener las aplicaciones activas para siempre y recargarlas sin tiempo de inactividad. PM2 también le permite administrar el registro de aplicaciones, la supervisión y la agrupación en clústeres.

Instale pm2 globalmente.

```
npm install -g pm2
```

Luego, ejecute la aplicación `node.js` usando PM2.

```
pm2 start server.js --name "my-app"
```

```
$ pm2 start app.js --name my-app  
[PM2] restartProcessId process id 0
```

App name	id	mode	pid	status	restart	uptime	memory	watching
my-app	0	fork	64029	online	1	0s	17.816 MB	disabled

Use the ``pm2 show <id|name>`` command to get more details about an app.

Los siguientes comandos son útiles mientras se trabaja con PM2.

Listar todos los procesos en ejecución:

```
pm2 list
```

Detener una aplicación:

```
pm2 stop my-app
```

Reinicie una aplicación:

```
pm2 restart my-app
```

Para ver información detallada sobre una aplicación:

```
pm2 show my-app
```

Para eliminar una aplicación del registro de PM2:

```
pm2 delete my-app
```

Despliegue usando el administrador de procesos

El administrador de procesos se usa generalmente en producción para implementar una aplicación nodejs. Las funciones principales de un administrador de procesos son reiniciar el servidor si falla, verificar el consumo de recursos, mejorar el rendimiento en tiempo de ejecución, monitorear, etc.

Algunos de los gestores de procesos populares creados por la comunidad de nodos son forever, pm2, etc.

Forever

`forever` es una herramienta de interfaz de línea de comandos para garantizar que un script dado se ejecute continuamente. La sencilla interfaz de `forever` lo hace ideal para ejecutar implementaciones más pequeñas de aplicaciones y scripts `Node.js`

`forever` supervisa su proceso y lo reinicia si se bloquea.

Instalar `forever` globalmente.

```
$ npm install -g forever
```

Ejecutar aplicación:

```
$ forever start server.js
```

Esto inicia el servidor y proporciona una identificación para el proceso (comienza desde 0).

Reiniciar aplicación :

```
$ forever restart 0
```

Aquí 0 es el id del servidor.

Detener la aplicación:

```
$ forever stop 0
```

Similar a reiniciar, 0 es el id del servidor. También puede dar el ID del proceso o el nombre del

script en lugar del ID dado por el siempre.

Para más comandos: <https://www.npmjs.com/package/forever>

Uso de diferentes propiedades / configuración para diferentes entornos como dev, qa, puesta en escena, etc.

Las aplicaciones a gran escala a menudo necesitan propiedades diferentes cuando se ejecutan en diferentes entornos. podemos lograrlo pasando argumentos a la aplicación NodeJs y usando el mismo argumento en el proceso del nodo para cargar un archivo de propiedades del entorno específico.

Supongamos que tenemos dos archivos de propiedades para diferentes entornos.

- dev.json

```
{
  "PORT": 3000,
  "DB": {
    "host": "localhost",
    "user": "bob",
    "password": "12345"
  }
}
```

- qa.json

```
{
  "PORT": 3001,
  "DB": {
    "host": "where_db_is_hosted",
    "user": "bob",
    "password": "54321"
  }
}
```

El siguiente código en la aplicación exportará el archivo de propiedad respectivo que queremos usar.

```
process.argv.forEach(function (val) {
  var arg = val.split("=");
  if (arg.length > 0) {
    if (arg[0] === 'env') {
      var env = require('./' + arg[1] + '.json');
      exports.prop = env;
    }
  }
});
```

Damos argumentos a la aplicación como sigue.

```
node app.js env=dev
```

Si estamos usando un administrador de procesos como *para siempre*, es tan simple como

```
forever start app.js env=dev
```

Aprovechando los clusters.

Una sola instancia de Node.js se ejecuta en un solo hilo. Para aprovechar los sistemas de múltiples núcleos, el usuario a veces querrá iniciar un clúster de procesos Node.js para manejar la carga.

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

var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // In real life, you'd probably use more than just 2 workers,
  // and perhaps not put the master and worker in the same file.
  //
  // You can also of course get a bit fancier about logging, and
  // implement whatever custom logic you need to prevent DoS
  // attacks and other bad behavior.
  //
  // See the options in the cluster documentation.
  //
  // The important thing is that the master does very little,
  // increasing our resilience to unexpected errors.
  console.log('your server is working on ' + numCPUs + ' cores');

  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('disconnect', function(worker) {
    console.error('disconnect!');
    //clearTimeout(timeout);
    cluster.fork();
  });
} else {
  require('./app.js');
}
```

Lea Despliegue de aplicaciones Node.js en producción en línea: <https://riptutorial.com/es/node-js/topic/2975/despliegue-de-aplicaciones-node-js-en-produccion>

Capítulo 34: Despliegue de la aplicación Node.js sin tiempo de inactividad.

Examples

Despliegue utilizando PM2 sin tiempo de inactividad.

ecosystem.json

```
{
  "name": "app-name",
  "script": "server",
  "exec_mode": "cluster",
  "instances": 0,
  "wait_ready": true
  "listen_timeout": 10000,
  "kill_timeout": 5000,
}
```

wait_ready

En lugar de volver a cargar esperando el evento de escucha, espere process.send ('ready');

listen_timeout

Tiempo en ms antes de forzar una recarga si la aplicación no escucha.

kill_timeout

Tiempo en ms antes de enviar un SIGKILL final.

server.js

```
const http = require('http');
const express = require('express');

const app = express();
const server = http.Server(app);
const port = 80;

server.listen(port, function() {
  process.send('ready');
});

process.on('SIGINT', function() {
  server.close(function() {
    process.exit(0);
  });
});
```

Es posible que deba esperar a que su aplicación haya establecido conexiones con sus DBs / caches / workers / lo que sea. PM2 debe esperar antes de considerar su solicitud como en línea. Para hacer esto, debe proporcionar `wait_ready: true` en un archivo de proceso. Esto hará que PM2 escuche ese evento. En su aplicación, deberá agregar `process.send('ready');` cuando quieras que tu aplicación sea considerada como lista.

Cuando PM2 detiene / reinicia un proceso, algunas señales del sistema se envían a su proceso en un orden determinado.

Primero se envía una señal `SIGINT` a sus procesos, señal que puede detectar para saber que su proceso se detendrá. Si su aplicación no sale sola antes de 1.6s (personalizable), recibirá una señal `SIGKILL` para forzar la salida del proceso. Entonces, si su aplicación necesita limpiar algunos estados o trabajos, puede detectar la señal `SIGINT` para preparar su aplicación para salir.

Lea [Despliegue de la aplicación Node.js sin tiempo de inactividad](https://riptutorial.com/es/node-js/topic/9752/despliegue-de-la-aplicacion-node-js-sin-tiempo-de-inactividad). en línea:

<https://riptutorial.com/es/node-js/topic/9752/despliegue-de-la-aplicacion-node-js-sin-tiempo-de-inactividad>

Capítulo 35: Devolución de llamada a la promesa

Examples

Prometiando una devolución de llamada

Basado en devolución de llamada:

```
db.notification.email.find({subject: 'promisify callback'}, (error, result) => {
  if (error) {
    console.log(error);
  }

  // normal code here
});
```

Esto utiliza el método `promisifyAll` de `bluebird` para promisifyar lo que es el código convencional basado en la devolución de llamada como el anterior. `bluebird` hará una versión prometidora de todos los métodos en el objeto, esos nombres de métodos basados en promesas tienen `Async` añadido a ellos:

```
let email = bluebird.promisifyAll(db.notification.email);

email.findAsync({subject: 'promisify callback'}).then(result => {

  // normal code here
})
.catch(console.error);
```

Si solo hay que prometer métodos específicos, solo use su `promisify`:

```
let find = bluebird.promisify(db.notification.email.find);

find({locationId: 168}).then(result => {

  // normal code here
});
.catch(console.error);
```

Hay algunas bibliotecas (por ejemplo, `MassiveJS`) que no se pueden prometer si el objeto inmediato del método no se pasa al segundo parámetro. En ese caso, simplemente pase el objeto inmediato del método que debe ser promisifyado en el segundo parámetro y encerrado en la propiedad de contexto.

```
let find = bluebird.promisify(db.notification.email.find, { context: db.notification.email });

find({locationId: 168}).then(result => {
```

```
    // normal code here
  });
  .catch(console.error);
```

Promisificando manualmente una devolución de llamada

A veces puede ser necesario promisificar manualmente una función de devolución de llamada. Esto podría ser en el caso de que la devolución de llamada no siga el [formato](#) estándar de [error primero](#) o si se necesita lógica adicional para prometer:

Ejemplo con [fs.exists \(ruta, callback\)](#) :

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

var existsAsync = function(path) {
  return new Promise(function(resolve, reject) {
    fs.exists(path, function(exists) {
      // exists is a boolean
      if (exists) {
        // Resolve successfully
        resolve();
      } else {
        // Reject with error
        reject(new Error('path does not exist'));
      }
    });
  });
};

// Use as a promise now
existsAsync('/path/to/some/file').then(function() {
  console.log('file exists!');
}).catch(function(err) {
  // file does not exist
  console.error(err);
});
```

setTimeout promisificado

```
function wait(ms) {
  return new Promise(function (resolve, reject) {
    setTimeout(resolve, ms)
  })
}
```

Lea Devolución de llamada a la promesa en línea: <https://riptutorial.com/es/node-js/topic/2346/devolucion-de-llamada-a-la-promesa>

Capítulo 36: Diseño API de descanso: Mejores prácticas

Examples

Manejo de errores: OBTENER todos los recursos

¿Cómo maneja los errores, en lugar de registrarlos en la consola?

Mal camino:

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        res.json(r)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    });
    request.info.user = req.user._id;
    console.log("ABOUT TO SAVE REQUEST", request);
    request.save((err, r) => {
      if (err) {
        res.json({ message: 'there was an error saving your r' });
      } else {
        res.json(r);
      }
    });
  });
});
```

Mejor manera:

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        return next(err)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    })
```

```
});  
request.info.user = req.user._id;  
console.log("ABOUT TO SAVE REQUEST", request);  
request.save((err, r) => {  
  if (err) {  
    return next(err)  
  } else {  
    res.json(r);  
  }  
});  
});
```

Lea Diseño API de descanso: Mejores prácticas en línea: <https://riptutorial.com/es/node-js/topic/6490/disen%C3%B3-api-de-descanso--mejores-pr%C3%A1cticas>

Capítulo 37: ECMAScript 2015 (ES6) con Node.js

Examples

const / let declaraciones

A diferencia de `var`, `const` / `let` están vinculados al ámbito léxico en lugar del ámbito de la función.

```
{
  var x = 1 // will escape the scope
  let y = 2 // bound to lexical scope
  const z = 3 // bound to lexical scope, constant
}

console.log(x) // 1
console.log(y) // ReferenceError: y is not defined
console.log(z) // ReferenceError: z is not defined
```

[Ejecutar en RunKit](#)

Funciones de flecha

Las funciones de flecha se enlazan automáticamente al 'este' ámbito léxico del código circundante.

```
performSomething(result => {
  this.someVariable = result
})
```

VS

```
performSomething(function(result) {
  this.someVariable = result
}.bind(this))
```

Ejemplo de función de flecha

Consideremos este ejemplo, que muestra los cuadrados de los números 3, 5 y 7:

```
let nums = [3, 5, 7]
let squares = nums.map(function (n) {
  return n * n
})
console.log(squares)
```

[Ejecutar en RunKit](#)

La función pasada a `.map` también se puede escribir como función de flecha eliminando la palabra clave de la `function` y, en su lugar, agregando la flecha `=>` :

```
let nums = [3, 5, 7]
let squares = nums.map((n) => {
  return n * n
})
console.log(squares)
```

Ejecutar en RunKit

Sin embargo, esto puede ser escrito aún más conciso. Si el cuerpo de la función consta de una sola instrucción y esa declaración calcula el valor de retorno, se pueden eliminar las llaves de la envoltura del cuerpo de la función, así como la palabra clave `return` .

```
let nums = [3, 5, 7]
let squares = nums.map(n => n * n)
console.log(squares)
```

Ejecutar en RunKit

desestructuración

```
let [x,y, ...nums] = [0, 1, 2, 3, 4, 5, 6];
console.log(x, y, nums);

let {a, b, ...props} = {a:1, b:2, c:3, d:{e:4}}
console.log(a, b, props);

let dog = {name: 'fido', age: 3};
let {name:n, age} = dog;
console.log(n, age);
```

fluir

```
/* @flow */

function product(a: number, b: number){
  return a * b;
}

const b = 3;
let c = [1,2,3,,{}];
let d = 3;

import request from 'request';

request('http://dev.markitondemand.com/MODApis/Api/v2/Quote/json?symbol=AAPL', (err, res,
payload)=>{
  payload = JSON.parse(payload);
  let {LastPrice} = payload;
  console.log(LastPrice);
});
```

Clase ES6

```
class Mammel {
  constructor(legs) {
    this.legs = legs;
  }
  eat() {
    console.log('eating...');
  }
  static count() {
    console.log('static count...');
  }
}

class Dog extends Mammel {
  constructor(name, legs) {
    super(legs);
    this.name = name;
  }
  sleep() {
    super.eat();
    console.log('sleeping');
  }
}

let d = new Dog('fido', 4);
d.sleep();
d.eat();
console.log('d', d);
```

Lea ECMAScript 2015 (ES6) con Node.js en línea: <https://riptutorial.com/es/node-js/topic/6732/ecmascript-2015--es6--con-node-js>

Capítulo 38: Ejecutando archivos o comandos con procesos hijo

Sintaxis

- `child_process.exec` (comando [, opciones] [, devolución de llamada])
- `child_process.execFile` (archivo [, argumentos] [, opciones] [, devolución de llamada])
- `child_process.fork` (modulePath [, args] [, opciones])
- `child_process.spawn` (comando [, args] [, opciones])
- `child_process.execFileSync` (archivo [, argumentos] [, opciones])
- `child_process.execSync` (comando [, opciones])
- `child_process.spawnSync` (comando [, args] [, opciones])

Observaciones

Cuando se trata de procesos secundarios, todos los métodos asíncronos devolverán una instancia de `ChildProcess`, mientras que todas las versiones síncronas devolverán la salida de lo que se haya ejecutado. Al igual que otras operaciones síncronas en Node.js, si se produce un error, se *tirará*.

Examples

Generando un nuevo proceso para ejecutar un comando.

Para generar un nuevo proceso en el que necesite una salida *sin almacenamiento* (por ejemplo, procesos de larga ejecución que puedan imprimir la salida durante un período de tiempo en lugar de imprimir y salir de inmediato), use `child_process.spawn()`.

Este método genera un nuevo proceso utilizando un comando dado y una matriz de argumentos. El valor de retorno es una instancia de `ChildProcess`, que a su vez proporciona las propiedades `stdout` y `stderr`. Ambas secuencias son instancias de `stream.Readable`.

El siguiente código es equivalente a usar la ejecución del comando `ls -lh /usr`.

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
```

```
console.log(`child process exited with code ${code}`);
});
```

Otro comando de ejemplo:

```
zip -0vr "archive" ./image.png
```

Podría escribirse como:

```
spawn('zip', ['-0vr', '"archive"', './image.png']);
```

Generando un shell para ejecutar un comando.

Para ejecutar un comando en un shell, en el que requería una salida con búfer (es decir, no es una secuencia), use `child_process.exec`. Por ejemplo, si desea ejecutar el comando `cat *.js file | wc -l`, sin opciones, se vería así:

```
const exec = require('child_process').exec;
exec('cat *.js file | wc -l', (err, stdout, stderr) => {
  if (err) {
    console.error(`exec error: ${err}`);
    return;
  }

  console.log(`stdout: ${stdout}`);
  console.log(`stderr: ${stderr}`);
});
```

La función acepta hasta tres parámetros:

```
child_process.exec(command[, options][, callback]);
```

El parámetro de comando es una cadena, y es obligatorio, mientras que el objeto de opciones y la devolución de llamada son opcionales. Si no se especifica ningún objeto de opciones, `exec` usará lo siguiente como predeterminado:

```
{
  encoding: 'utf8',
  timeout: 0,
  maxBuffer: 200*1024,
  killSignal: 'SIGTERM',
  cwd: null,
  env: null
}
```

El objeto de opciones también admite un parámetro de `shell`, que es de forma predeterminada `/bin/sh` en UNIX y `cmd.exe` en Windows, una opción `uid` para configurar la identidad del usuario del proceso y una opción `gid` para la identidad del grupo.

La devolución de llamada, que se invoca cuando se termina de ejecutar el comando, se llama con

los tres argumentos (`err`, `stdout`, `stderr`) . Si el comando se ejecuta con éxito, `err` será `null` , de lo contrario será una instancia de `Error` , donde `err.code` será el código de salida del proceso y `err.signal` será la señal que se envió para finalizarlo.

Los argumentos `stdout` y `stderr` son la salida del comando. Se decodifica con la codificación especificada en el objeto de opciones (predeterminado: `string`), pero de lo contrario se puede devolver como un objeto `Buffer` .

También existe una versión síncrona de `exec` , que es `execSync` . La versión síncrona no recibe una devolución de llamada y devolverá `stdout` lugar de una instancia de `ChildProcess` . Si la versión síncrona encuentra un error, se va a tirar y poner fin a su programa. Se parece a esto:

```
const execSync = require('child_process').execSync;
const stdout = execSync('cat *.js file | wc -l');
console.log(`stdout: ${stdout}`);
```

Generando un proceso para ejecutar un ejecutable.

Si está buscando ejecutar un archivo, como un ejecutable, use `child_process.execFile` . En lugar de generar un shell como `child_process.exec` , creará directamente un nuevo proceso, que es un poco más eficiente que ejecutar un comando. La función se puede utilizar como tal:

```
const execFile = require('child_process').execFile;
const child = execFile('node', ['--version'], (err, stdout, stderr) => {
  if (err) {
    throw err;
  }

  console.log(stdout);
});
```

A diferencia de `child_process.exec` , esta función aceptará hasta cuatro parámetros, donde el segundo parámetro es una matriz de argumentos que le gustaría proporcionar al ejecutable:

```
child_process.execFile(file[, args[, options[, callback]]];
```

De lo contrario, las opciones y el formato de devolución de llamada son idénticos a `child_process.exec` . Lo mismo ocurre con la versión síncrona de la función:

```
const execFileSync = require('child_process').execFileSync;
const stdout = execFileSync('node', ['--version']);
console.log(stdout);
```

Lea Ejecutando archivos o comandos con procesos hijo en línea: <https://riptutorial.com/es/node-js/topic/2726/ejecutando-archivos-o-comandos-con-procesos-hijo>

Capítulo 39: Ejecutando node.js como un servicio

Introducción

A diferencia de muchos servidores web, Node no se instala como un servicio fuera de la caja. Pero en producción, es mejor tenerlo ejecutado como un demonio, gestionado por un sistema init.

Examples

Node.js como un sistema de demonio

systemd es el sistema inicial *de facto* en la mayoría de las distribuciones de Linux. Después de que Node se haya configurado para ejecutarse con systemd, es posible usar el comando de `service` para administrarlo.

En primer lugar, necesita un archivo de configuración, vamos a crearlo. Para las distribuciones basadas en Debian, estará en `/etc/systemd/system/node.service`

```
[Unit]
Description=My super nodejs app

[Service]
# set the working directory to have consistent relative paths
WorkingDirectory=/var/www/app

# start the server file (file is relative to WorkingDirectory here)
ExecStart=/usr/bin/node serverCluster.js

# if process crashes, always try to restart
Restart=always

# let 500ms between the crash and the restart
RestartSec=500ms

# send log tot syslog here (it doesn't compete with other log config in the app itself)
StandardOutput=syslog
StandardError=syslog

# nodejs process name in syslog
SyslogIdentifier=nodejs

# user and group starting the app
User=www-data
Group=www-data

# set the environement (dev, prod...)
Environment=NODE_ENV=production

[Install]
```

```
# start node at multi user system level (= sysVinit runlevel 3)
WantedBy=multi-user.target
```

Ahora es posible iniciar, detener y reiniciar la aplicación respectivamente con:

```
service node start
service node stop
service node restart
```

Para indicarle a systemd que inicie automáticamente el nodo en el arranque, simplemente escriba: `systemctl enable node`.

Eso es todo, el nodo ahora se ejecuta como un demonio.

Lea Ejecutando node.js como un servicio en línea: <https://riptutorial.com/es/node-js/topic/9258/ejecutando-node-js-como-un-servicio>

Capítulo 40: Emisores de eventos

Observaciones

Cuando un evento se "dispara" (lo que significa lo mismo que "publicar un evento" o "emitir un evento"), cada oyente se llamará de forma sincrónica ([fuente](#)), junto con los datos adjuntos que se pasaron a `emit()` , no importa cuántos argumentos pases en:

```
myDog.on('bark', (howLoud, howLong, howIntense) => {
  // handle the event
})
myDog.emit('bark', 'loudly', '5 seconds long', 'fiercely')
```

Los oyentes serán llamados en el orden en que fueron registrados:

```
myDog.on('urinate', () => console.log('My first thought was "Oh-no"'))
myDog.on('urinate', () => console.log('My second thought was "Not my lawn :)"))
myDog.emit('urinate')
// The console.logs will happen in the right order because they were registered in that order.
```

Pero si necesita un oyente para disparar primero, antes de todos los otros oyentes que ya se han agregado, puede usar `prependListener()` así:

```
myDog.prependListener('urinate', () => console.log('This happens before my first and second
thoughts, even though it was registered after them'))
```

Si necesita escuchar un evento, pero solo desea escucharlo una vez, puede usarlo `once` vez `on` lugar de `on` , o `prependOnceListener` lugar de `prependListener` . Una vez que se activa el evento y se llama al oyente, éste se eliminará automáticamente y no se volverá a llamar la próxima vez que se active el evento.

Finalmente, si desea eliminar a todos los oyentes y comenzar de nuevo, siéntase libre de hacer eso:

```
myDog.removeAllListeners()
```

Examples

HTTP Analytics a través de un emisor de eventos

En el código del servidor HTTP (por ejemplo, `server.js`):

```
const EventEmitter = require('events')
const serverEvents = new EventEmitter()

// Set up an HTTP server
const http = require('http')
```

```
const httpServer = http.createServer((request, response) => {
  // Handler the request...
  // Then emit an event about what happened
  serverEvents.emit('request', request.method, request.url)
});

// Expose the event emitter
module.exports = serverEvents
```

En código de supervisor (ej. `supervisor.js`):

```
const server = require('./server.js')
// Since the server exported an event emitter, we can listen to it for changes:
server.on('request', (method, url) => {
  console.log(`Got a request: ${method} ${url}`)
})
```

Cada vez que el servidor recibe una solicitud, emitirá un evento llamado `request` que el supervisor está escuchando, y luego el supervisor puede reaccionar al evento.

Lo esencial

Los emisores de eventos están integrados en Node, y son para pub-sub, un patrón donde un *editor* emitirá eventos, a los que los *suscriptores* pueden escuchar y reaccionar. En la jerga de Nodos, los editores se denominan *Emisores de eventos* y emiten eventos, mientras que los suscriptores se llaman *escuchas* y reaccionan a los eventos.

```
// Require events to start using them
const EventEmitter = require('events').EventEmitter;
// Dogs have events to publish, or emit
class Dog extends EventEmitter {};
class Food {};

let myDog = new Dog();

// When myDog is chewing, run the following function
myDog.on('chew', (item) => {
  if (item instanceof Food) {
    console.log('Good dog');
  } else {
    console.log(`Time to buy another ${item}`);
  }
});

myDog.emit('chew', 'shoe'); // Will result in console.log('Time to buy another shoe')
const bacon = new Food();
myDog.emit('chew', bacon); // Will result in console.log('Good dog')
```

En el ejemplo anterior, el perro es el editor / `EventEmitter`, mientras que la función que verifica el elemento fue el suscriptor / oyente. Puedes hacer más oyentes también:

```
myDog.on('bark', () => {
  console.log('WHO'S AT THE DOOR?');
  // Panic
```

```
});
```

También puede haber múltiples escuchas para un solo evento, e incluso eliminar escuchas:

```
myDog.on('chew', takeADeepBreathe);
myDog.on('chew', calmDown);
// Undo the previous line with the next one:
myDog.removeListener('chew', calmDown);
```

Si desea escuchar un evento solo una vez, puede utilizar:

```
myDog.once('chew', pet);
```

Lo que eliminará al oyente automáticamente sin condiciones de carrera.

Obtenga los nombres de los eventos a los que está suscrito.

La función **EventEmitter.eventNames ()** devolverá una matriz que contiene los nombres de los eventos a los que está suscrito actualmente.

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}

var emitter = new MyEmitter();

emitter
.on("message", function(){ //listen for message event
  console.log("a message was emitted!");
})
.on("message", function(){ //listen for message event
  console.log("this is not the right message");
})
.on("data", function(){ //listen for data event
  console.log("a data just occurred!!");
});

console.log(emitter.eventNames()); //=> ["message","data"]
emitter.removeAllListeners("data");//=> removeAllListeners to data event
console.log(emitter.eventNames()); //=> ["message"]
```

Ejecutar en RunKit

Obtenga el número de oyentes registrados para escuchar un evento específico

La función **Emitter.listenerCount (eventName)** devolverá el número de escuchas que están escuchando actualmente el evento proporcionado como argumento

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}
var emitter = new MyEmitter();
```

```
emitter
.on("data", ()=>{ // add listener for data event
  console.log("data event emitter");
});

console.log(emitter.listenerCount("data"))    // => 1
console.log(emitter.listenerCount("message")) // => 0

emitter.on("message", function mListener(){ //add listener for message event
  console.log("message event emitted");
});
console.log(emitter.listenerCount("data"))    // => 1
console.log(emitter.listenerCount("message")) // => 1

emitter.once("data", (stuff)=>{ //add another listener for data event
  console.log(`Tell me my ${stuff}`);
})

console.log(emitter.listenerCount("data"))    // => 2
console.log(emitter.listenerCount("message")) // => 1
```

Lea Emisores de eventos en línea: <https://riptutorial.com/es/node-js/topic/1623/emisores-de-eventos>

Capítulo 41: Enrutamiento de solicitudes ajax con Express.JS

Examples

Una implementación sencilla de AJAX.

Debes tener la plantilla básica de generador express

En app.js, agregue (puede agregarlo en cualquier lugar después de `var app = express.app()`):

```
app.post(function(req, res, next){
  next();
});
```

Ahora en su archivo index.js (o su respectiva coincidencia), agregue:

```
router.get('/ajax', function(req, res){
  res.render('ajax', {title: 'An Ajax Example', quote: "AJAX is great!"});
});
router.post('/ajax', function(req, res){
  res.render('ajax', {title: 'An Ajax Example', quote: req.body.quote});
});
```

Cree un `ajax.jade` / `ajax.pug` o `ajax.ejs` en el directorio `/views`, agregue:

Para Jade / PugJS:

```
extends layout
script(src="http://code.jquery.com/jquery-3.1.0.min.js")
script(src="/magic.js")
h1 Quote: #{quote}
form(method="post" id="changeQuote")
  input(type='text', placeholder='Set quote of the day', name='quote')
  input(type="submit", value="Save")
```

Para EJS:

```
<script src="http://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="/magic.js"></script>
<h1>Quote: <%=quote%> </h1>
<form method="post" id="changeQuote">
  <input type="text" placeholder="Set quote of the day" name="quote"/>
  <input type="submit" value="Save">
</form>
```

Ahora, crea un archivo en `/public` llamado `magic.js`

```
$(document).ready(function(){
```

```
$("#form#changeQuote").on('submit', function(e){
    e.preventDefault();
    var data = $('#input[name=quote]').val();
    $.ajax({
        type: 'post',
        url: '/ajax',
        data: data,
        dataType: 'text'
    })
    .done(function(data){
        $('#h1').html(data.quote);
    });
});
```

¡Y ahí lo tienes! Al hacer clic en Guardar, la cotización cambiará!

Lea Enrutamiento de solicitudes ajax con Express.JS en línea: <https://riptutorial.com/es/node-js/topic/6738/enrutamiento-de-solicitudes-ajax-con-express-js>

Capítulo 42: Enrutamiento NodeJs

Introducción

Cómo configurar un servidor web Express básico bajo el nodo js y Explorando el enrutador Express.

Observaciones

Por último, utilizando Express Router puede usar la facilidad de enrutamiento en su aplicación y es fácil de implementar.

Examples

Enrutamiento de Express Web Server

Creación de Express Web Server

El servidor Express fue práctico y profundiza en muchos usuarios y comunidades. Se está volviendo popular.

Permite crear un Servidor Express. Para la administración de paquetes y la flexibilidad para la dependencia utilizaremos NPM (Administrador de paquetes de nodos).

1. Vaya al directorio Proyecto y cree el archivo package.json. **package.json** {"name": "expressRouter", "version": "0.0.1", "scripts": {"start": "node Server.js"}, "dependencies": {"express": "^ 4.12.3 "}}
2. Guarde el archivo e instale la dependencia expresa utilizando el siguiente comando *npm install* . Esto creará node_modules en su directorio de proyecto junto con la dependencia requerida.
3. Vamos a crear Express Web Server. Vaya al directorio Proyecto y cree el archivo server.js. **server.js**

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

```
// Creando el objeto Router ()
```

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

```
// Proporcionar todas las rutas aquí, esto es para la página de inicio.
```

```
router.get ("/", function (req, res) {  
  res.json({"message" : "Hello World"});  
});
```

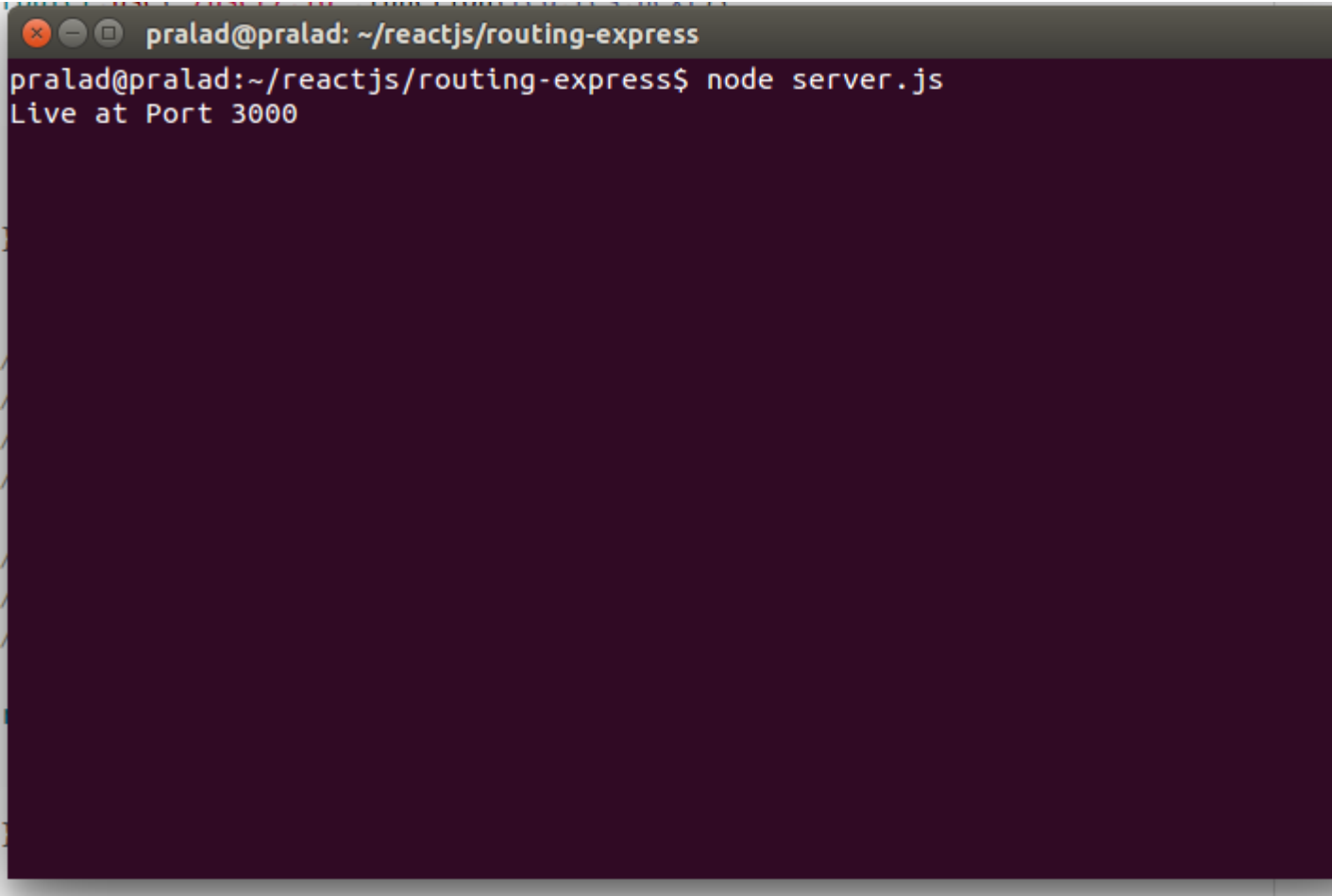
```
});  
  
app.use ("/ api", enrutador);  
  
// Escucha este puerto  
  
app.listen (3000, function () {console.log ("Live at Port 3000");});
```

For more detail on setting node server you can see [\[here\]](#)[1].

4. Ejecuta el servidor escribiendo el siguiente comando.

```
nodo server.js
```

Si el servidor se ejecuta correctamente, verás algo como esto.

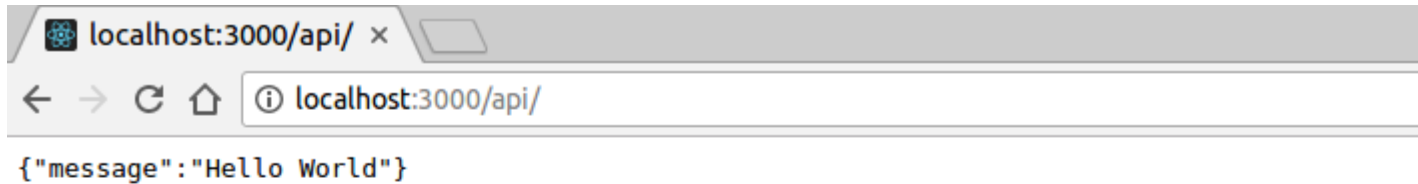
A screenshot of a terminal window with a dark purple background. The window title is 'pralad@pralad: ~/reactjs/routing-express'. The prompt is 'pralad@pralad:~/reactjs/routing-express\$'. The command 'node server.js' has been entered, and the output 'Live at Port 3000' is displayed on the next line.

```
pralad@pralad: ~/reactjs/routing-express  
pralad@pralad:~/reactjs/routing-express$ node server.js  
Live at Port 3000
```

5. Ahora ve al navegador o al cartero y haz una solicitud.

[http: // localhost: 3000 / api /](http://localhost:3000/api/)

La salida será



Eso es todo, lo básico del enrutamiento Express.

Ahora vamos a manejar el GET, POST etc.

Cambiar your server.js archivo como

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

//Creating Router() object

var router = express.Router();

// Router middleware, mentioned it before defining routes.

router.use(function(req,res,next) {
  console.log("/") + req.method;
  next();
});

// Provide all routes here, this is for Home page.

router.get("/",function(req,res){
  res.json({"message" : "Hello World"});
});

app.use("/api",router);

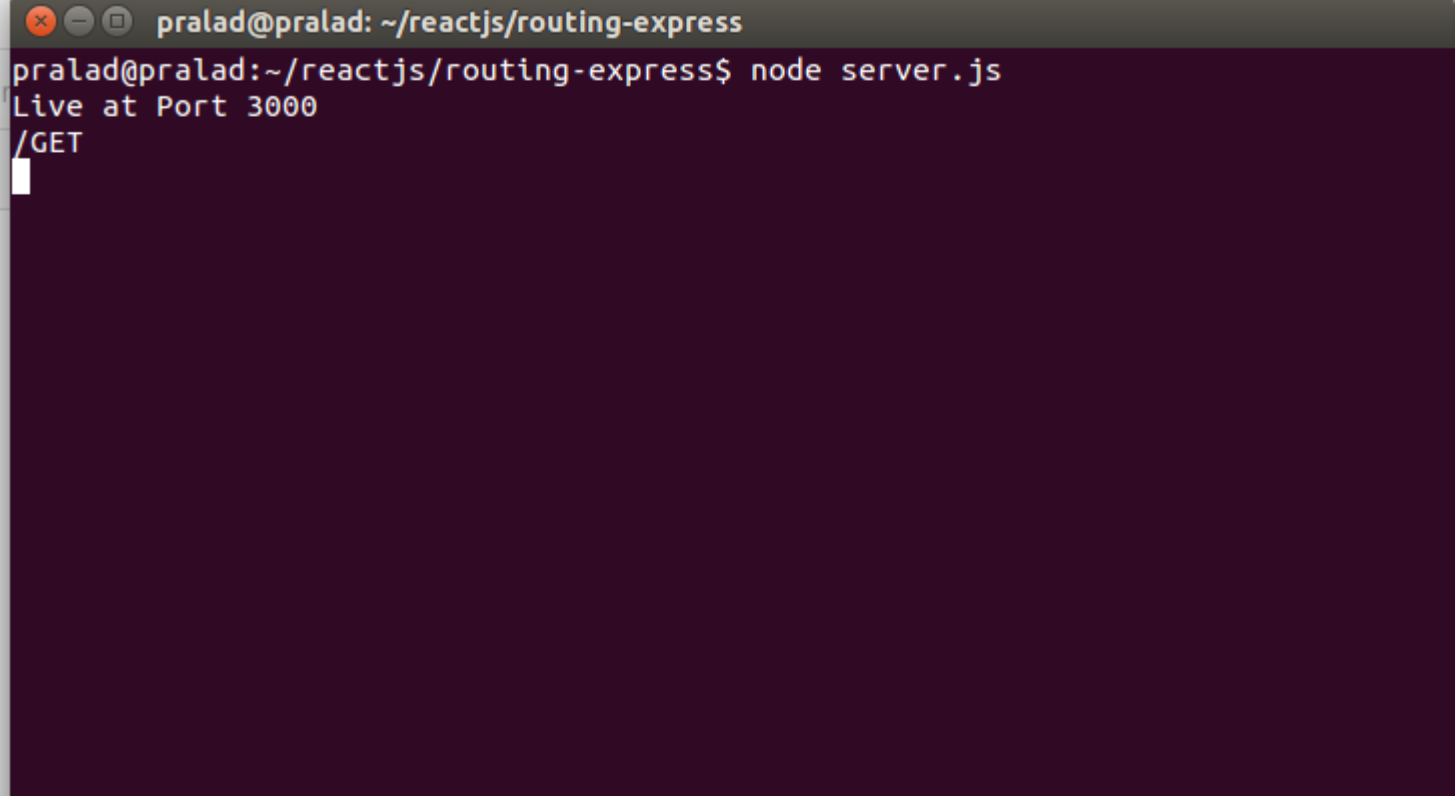
app.listen(3000,function(){
  console.log("Live at Port 3000");
});
```

```
});
```

Ahora si reinicia el servidor y realiza la solicitud a

```
http://localhost:3000/api/
```

Verás algo como



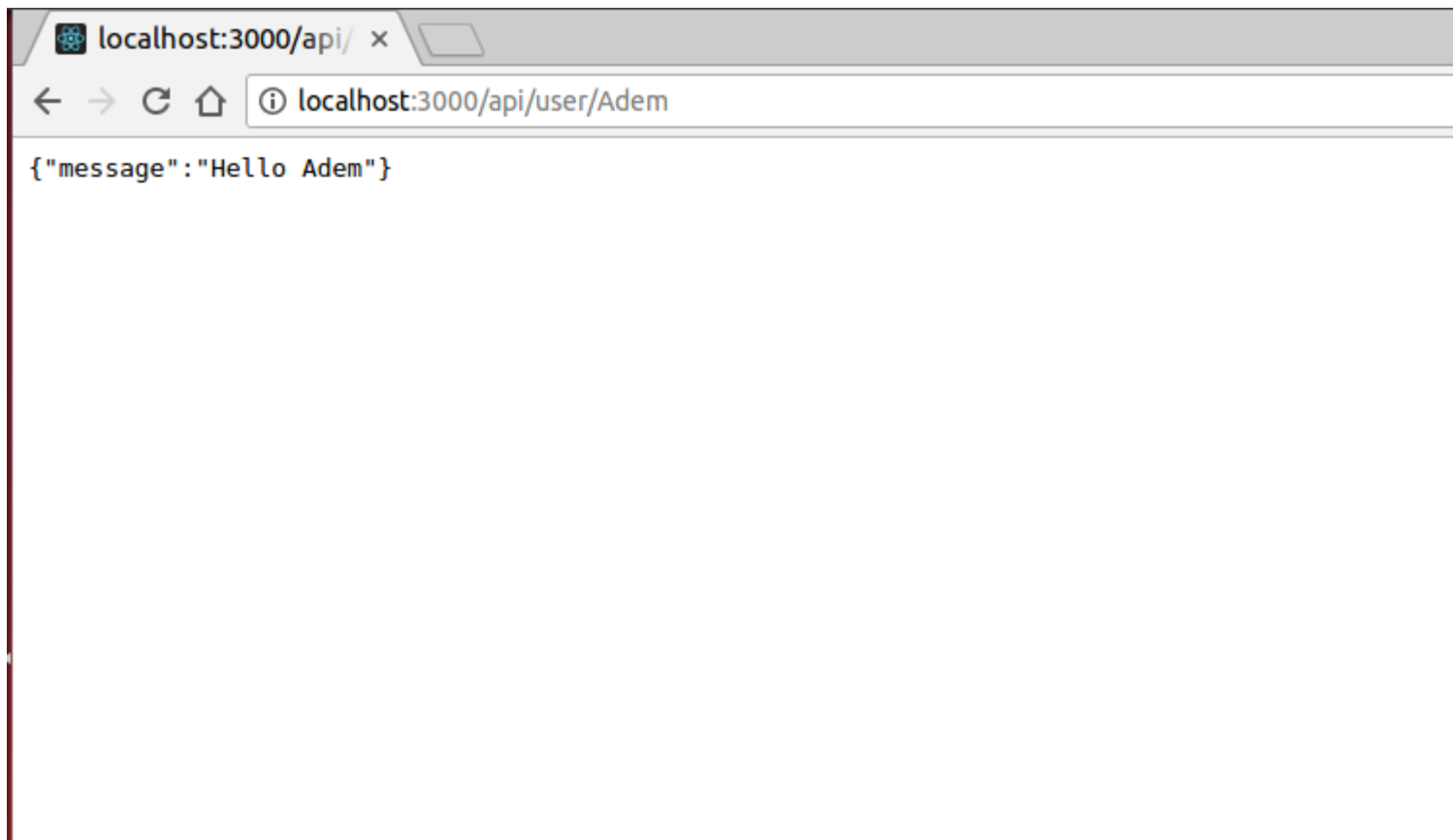
```
pralad@pralad: ~/reactjs/routing-express
pralad@pralad:~/reactjs/routing-express$ node server.js
Live at Port 3000
/GET
```

Parámetros de acceso en el enrutamiento

También puede acceder al parámetro desde url, como <http://example.com/api/:name/> . Así se puede acceder al parámetro nombre. Agregue el siguiente código en su server.js

```
router.get("/user/:id", function(req, res) {
  res.json({"message" : "Hello "+req.params.id});
});
```

Ahora reinicie el servidor y vaya a [<http://localhost:3000/api/user/Adem>] [4] , la salida será como



Lea Enrutamiento NodeJs en línea: <https://riptutorial.com/es/node-js/topic/9846/enrutamiento-nodejs>

Capítulo 43: Entregar HTML o cualquier otro tipo de archivo.

Sintaxis

- `response.sendFile (fileName, options, function (err) {});`

Examples

Entregar HTML en la ruta especificada

Aquí se explica cómo crear un servidor Express y servir `index.html` de forma predeterminada (ruta vacía `/`) y `page1.html` para la ruta `/page1`.

Estructura de la carpeta

```
project root
|   server.js
|___views
|   index.html
|   page1.html
```

server.js

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

// deliver index.html if no file is requested
app.get("/", function (request, response) {
  response.sendFile(path.join(__dirname, 'views/index.html'));
});

// deliver page1.html if page1 is requested
app.get('/page1', function(request, response) {
  response.sendFile(path.join(__dirname, 'views', 'page1.html', function(error) {
    if (error) {
      // do something in case of error
      console.log(err);
      response.end(JSON.stringify({error:"page not found"}));
    }
  }));
});

app.listen(8080);
```

Tenga en cuenta que `sendFile()` simplemente transmite un archivo estático como respuesta, sin

ofrecer la oportunidad de modificarlo. Si está sirviendo un archivo HTML y desea incluir datos dinámicos con él, entonces necesitará usar un *motor de plantillas* como Pug, Moustache o EJS.

Lea *Entregar HTML o cualquier otro tipo de archivo. en línea*: <https://riptutorial.com/es/node-js/topic/6538/entregar-html-o-cualquier-otro-tipo-de-archivo->

Capítulo 44: Enviando un flujo de archivos al cliente

Examples

Uso de fs y pipe para transmitir archivos estáticos desde el servidor

Un buen servicio de VOD (Video On Demand) debe comenzar con lo básico. Digamos que tiene un directorio en su servidor al que no se puede acceder públicamente, pero a través de algún tipo de portal o muro de paja desea que los usuarios accedan a sus medios.

```
var movie = path.resolve('./public/' + req.params.filename);

fs.stat(movie, function (err, stats) {

  var range = req.headers.range;

  if (!range) {

    return res.sendStatus(416);

  }

  //Chunk logic here
  var positions = range.replace(/bytes=/, "").split("-");
  var start = parseInt(positions[0], 10);
  var total = stats.size;
  var end = positions[1] ? parseInt(positions[1], 10) : total - 1;
  var chunksize = (end - start) + 1;

  res.writeHead(206, {

    'Transfer-Encoding': 'chunked',

    "Content-Range": "bytes " + start + "-" + end + "/" + total,

    "Accept-Ranges": "bytes",

    "Content-Length": chunksize,

    "Content-Type": mime.lookup(req.params.filename)

  });

  var stream = fs.createReadStream(movie, { start: start, end: end, autoClose: true

  });

  .on('end', function () {

    console.log('Stream Done');

  })

  .on("error", function (err) {
```

```
        res.end(err);

    })

    .pipe(res, { end: true });

});
```

El fragmento anterior es un resumen básico de cómo le gustaría transmitir su video a un cliente. La lógica del fragmento depende de una variedad de factores, incluido el tráfico de red y la latencia. Es importante equilibrar el tamaño del mandril frente a la cantidad.

Finalmente, la llamada `.pipe` le permite a `node.js` mantener una conexión abierta con el servidor y enviar fragmentos adicionales según sea necesario.

Streaming Utilizando `fluent-ffmpeg`

También puede usar `flent-ffmpeg` para convertir archivos `.mp4` a archivos `.flv` u otros tipos:

`res.contentType ('flv');`

```
var pathToMovie = './public/' + req.params.filename;

var proc = ffmpeg(pathToMovie)

    .preset('flashvideo')

    .on('end', function () {

        console.log('Stream Done');

    })

    .on('error', function (err) {

        console.log('an error happened: ' + err.message);

        res.send(err.message);

    })

    .pipe(res, { end: true });
```

Lea Enviando un flujo de archivos al cliente en línea: <https://riptutorial.com/es/node-js/topic/6994/enviando-un-flujo-de-archivos-al-cliente>

Capítulo 45: Enviar notificación web

Examples

Enviar notificación web utilizando GCM (Google Cloud Messaging System)

Dicho ejemplo es la difusión amplia entre aplicaciones **PWA** (aplicaciones web progresivas) y en este ejemplo enviaremos una simple notificación tipo Backend utilizando **NodeJS** y **ES6**

1. Instale el módulo Node-GCM: `npm install node-gcm`
2. Instale Socket.io: `npm install socket.io`
3. Cree una aplicación habilitada para GCM utilizando la [consola de Google](#).
4. Grabe su ID de aplicación GCM (la necesitaremos más adelante)
5. Grabe su código secreto de aplicación GCM.
6. Abra su editor de código favorito y agregue el siguiente código:

```
'use strict';

const express = require('express');
const app = express();
const gcm = require('node-gcm');
app.io = require('socket.io')();

// [*] Configuring our GCM Channel.
const sender = new gcm.Sender('Project Secret');
const regTokens = [];
let message = new gcm.Message({
  data: {
    key1: 'msg1'
  }
});

// [*] Configuring our static files.
app.use(express.static('public/'));

// [*] Configuring Routes.
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/public/index.html');
});

// [*] Configuring our Socket Connection.
app.io.on('connection', socket => {
  console.log('we have a new connection ...');
  socket.on('new_user', (reg_id) => {
    // [*] Adding our user notification registration token to our list typically
    // hided in a secret place.
    if (regTokens.indexOf(reg_id) === -1) {
      regTokens.push(reg_id);
    }
  });
});
```

```

        // [*] Sending our push messages
        sender.send(message, {
            registrationTokens: regTokens
        }, (err, response) => {
            if (err) console.error('err', err);
            else console.log(response);
        });
    }
})
});

module.exports = app

```

PD: Estoy usando aquí un truco especial para hacer que Socket.io funcione con Express porque simplemente no funciona fuera de la caja.

Ahora **crea** un archivo **.json** y **asígnale** el nombre: **Manifest.json** , ábralo y **pase** lo siguiente:

```

{
  "name": "Application Name",
  "gcm_sender_id": "GCM Project ID"
}

```

Ciérralo y guárdalo en el directorio **ROOT** de tu aplicación.

PD: el archivo Manifest.json debe estar en el directorio raíz o no funcionará.

En el código anterior estoy haciendo lo siguiente:

1. Configuré y envié una página index.html normal que también usará socket.io.
2. Estoy escuchando un evento de **conexión** activado desde el **front-end** también conocido como mi **página index.html** (se activará una vez que un nuevo cliente se conecte con éxito a nuestro enlace predefinido)
3. Estoy enviando un conocimiento de token especial como el **token de registro** de mi index.html a través del evento socket.io **new_user** ; dicho token será nuestro código de acceso exclusivo del usuario y cada código se genera generalmente desde un navegador compatible para la **API de notificación web** (lea más [aquí](#)).
4. Simplemente estoy usando el módulo **node-gcm** para enviar mi notificación, la cual se manejará y se mostrará más adelante usando **Service Workers** '.

Esto es desde el punto de vista de **NodeJS** . en otros ejemplos, mostraré cómo podemos enviar datos personalizados, íconos, etc., en nuestro mensaje push.

PD: puedes encontrar la demo completa [aquí](#).

Lea **Enviar notificación web en línea**: <https://riptutorial.com/es/node-js/topic/6333/enviar-notificacion-web>

Capítulo 46: Estructura del proyecto

Introducción

La estructura del proyecto nodejs está influenciada por las preferencias personales, la arquitectura del proyecto y la estrategia de inyección del módulo que se está utilizando. También en el arco basado en eventos, que utiliza el mecanismo dinámico de creación de instancias del módulo. Para tener una estructura MVC es imperativo separar el código fuente del lado del servidor y del lado del cliente, ya que el código del lado del cliente probablemente se minimizará y se enviará al navegador y es de carácter público. Y el lado del servidor o el back-end proporcionarán API para realizar operaciones CRUD

Observaciones

El proyecto anterior utiliza los módulos browserify y vue.js como vista de base de aplicaciones y librerías de minificación. Por lo tanto, la estructura del proyecto puede cambiar minuciosamente según el marco de mvc que use, por ejemplo, el directorio de compilación en público deberá contener todo el código de mvc. Puedes tener una tarea que haga esto por ti.

Examples

Una sencilla aplicación nodejs con MVC y API.

- La primera distinción importante es entre los directorios generados dinámicamente que se utilizarán para los directorios de alojamiento y de origen.
- Los directorios de origen tendrán un archivo o carpeta de configuración según la cantidad de configuración que pueda tener. Esto incluye la configuración del entorno y la configuración de la lógica empresarial que puede elegir colocar dentro del directorio de configuración.

```
|-- Config
    |-- config.json
    |-- appConfig
    |-- pets.config
    |-- payment.config
```

- Ahora los directorios más importantes donde distinguimos entre el lado del servidor / backend y los módulos frontend. El *servidor* de 2 directorios y la aplicación *web* representan el backend y el frontend, respectivamente, que podemos elegir colocar dentro de un directorio de origen a saber. *src* .

Puede elegir diferentes nombres según la elección personal para el servidor o la aplicación web, según lo que tenga sentido para usted. Asegúrese de no querer hacerlo demasiado largo o complejo, ya que se encuentra en la estructura interna del proyecto final.

- Dentro del directorio del *servidor* puede tener el controlador, App.js / index.js, que será su archivo principal de nodejs y el punto de inicio. El directorio del servidor. también puede tener el *dto* dir que contiene todos los objetos de transferencia de datos que serán utilizados por los controladores API.

```
|-- server
  |-- dto
    |-- pet.js
    |-- payment.js
  |-- controller
    |-- PetsController.js
    |-- PaymentController.js
  |-- App.js
```

- El directorio de la aplicación web se puede dividir en dos partes principales *públicas* y *mvc* , esto se ve influenciado nuevamente por la estrategia de compilación que desee utilizar. Estamos utilizando [browserfiy](#) la compilación de la parte MVC de la aplicación web y minimizamos los contenidos del directorio *mvc* .

| - webapp | - public | - mvc

- Ahora el directorio público puede contener todos los recursos estáticos, imágenes, css (también puede tener archivos saas) y, lo más importante, los archivos HTML.

```
|-- public
  |-- build // will contained minified scripts(mvc)
  |-- images
    |-- mouse.jpg
    |-- cat.jpg
  |-- styles
    |-- style.css
  |-- views
    |-- petStore.html
    |-- paymentGateway.html
    |-- header.html
    |-- footer.html
  |-- index.html
```

- El directorio *mvc* contendrá la lógica de front-end, incluidos los *modelos* , los *controladores de vista* y cualquier otro módulo de *utils* que pueda necesitar como parte de la interfaz de usuario. También el index.js o shell.js, cualquiera que sea el conjunto de aplicaciones, también forma parte de este directorio.

```
|-- mvc
  |-- controllers
    |-- Dashborad.js
    |-- Help.js
    |-- Login.js
  |-- utils
  |-- index.js
```

Entonces, en conclusión, toda la estructura del proyecto se verá a continuación. Y una simple tarea de compilación como *gulp browserify minimizará* los scripts mvc y los publicará en un

directorio *público* . Luego podemos proporcionar este directorio público como recurso estático a través de la **api *express.use (satic ('public'))*** .

```
|-- node_modules
|-- src
|   |-- server
|       |-- controller
|       |-- App.js    // node app
|-- webapp
|   |-- public
|       |-- styles
|       |-- images
|       |-- index.html
|   |-- mvc
|       |-- controller
|       |-- shell.js  // mvc shell
|-- config
|-- Readme.md
|-- .gitignore
|-- package.json
```

Lea Estructura del proyecto en línea: <https://riptutorial.com/es/node-js/topic/9935/estructura-del-proyecto>

Capítulo 47: Eventloop

Introducción

En este post vamos a discutir cómo surgió el concepto de Eventloop y cómo se puede usar para servidores de alto rendimiento y aplicaciones controladas por eventos como las GUI.

Examples

Cómo evolucionó el concepto de bucle de eventos.

Eventloop en pseudo código

Un bucle de eventos es un bucle que espera eventos y luego reacciona a esos eventos

```
while true:
    wait for something to happen
    react to whatever happened
```

Ejemplo de un servidor HTTP de un solo hilo sin bucle de eventos

```
while true:
    socket = wait for the next TCP connection
    read the HTTP request headers from (socket)
    file_contents = fetch the requested file from disk
    write the HTTP response headers to (socket)
    write the (file_contents) to (socket)
    close(socket)
```

Aquí hay una forma simple de un servidor HTTP que es un solo hilo pero no un bucle de eventos. El problema aquí es que espera hasta que finalice cada solicitud antes de comenzar a procesar la siguiente. Si toma un tiempo leer los encabezados de solicitud HTTP o recuperar el archivo del disco, deberíamos poder comenzar a procesar la siguiente solicitud mientras esperamos que termine.

La solución más común es hacer que el programa sea multihilo.

Ejemplo de un servidor HTTP multihilo sin bucle de eventos

```

function handle_connection(socket):
    read the HTTP request headers from (socket)
    file_contents = fetch the requested file from disk
    write the HTTP response headers to (socket)
    write the (file_contents) to (socket)
    close(socket)
while true:
    socket = wait for the next TCP connection
    spawn a new thread doing handle_connection(socket)

```

Ahora hemos hecho nuestro pequeño servidor HTTP multihilo. De esta manera, podemos pasar inmediatamente a la siguiente solicitud porque la solicitud actual se está ejecutando en un subproceso en segundo plano. Muchos servidores, incluido Apache, utilizan este enfoque.

Pero no es perfecto. Una limitación es que solo puedes generar tantos hilos. Para las cargas de trabajo donde tiene una gran cantidad de conexiones, pero cada conexión solo requiere atención de vez en cuando, el modelo de subprocesos múltiples no funcionará muy bien. La solución para esos casos es usar un bucle de eventos:

Ejemplo de un servidor HTTP con bucle de eventos

```

while true:
    event = wait for the next event to happen
    if (event.type == NEW_TCP_CONNECTION):
        conn = new Connection
        conn.socket = event.socket
        start reading HTTP request headers from (conn.socket) with userdata = (conn)
    else if (event.type == FINISHED_READING_FROM_SOCKET):
        conn = event.userdata
        start fetching the requested file from disk with userdata = (conn)
    else if (event.type == FINISHED_READING_FROM_DISK):
        conn = event.userdata
        conn.file_contents = the data we fetched from disk
        conn.current_state = "writing headers"
        start writing the HTTP response headers to (conn.socket) with userdata = (conn)
    else if (event.type == FINISHED_WRITING_TO_SOCKET):
        conn = event.userdata
        if (conn.current_state == "writing headers"):
            conn.current_state = "writing file contents"
            start writing (conn.file_contents) to (conn.socket) with userdata = (conn)
        else if (conn.current_state == "writing file contents"):
            close(conn.socket)

```

Esperemos que este pseudocódigo sea inteligible. Esto es lo que está pasando: esperamos que las cosas sucedan. Cada vez que se crea una nueva conexión o una conexión existente necesita nuestra atención, nos ocupamos de ella y luego volvemos a esperar. De esa manera, nos desempeñamos bien cuando hay muchas conexiones y cada una rara vez requiere atención.

En una aplicación real (no pseudocódigo) que se ejecuta en Linux, la parte de "esperar a que ocurra el próximo evento" se implementaría llamando a la llamada del sistema `poll()` o `epoll()`.

Las partes de "comenzar a leer / escribir algo en un socket" se implementarían llamando a `recv ()` o a las llamadas del sistema `send ()` en modo no bloqueante.

Referencia:

[1]. "¿Cómo funciona un bucle de eventos?" [En línea]. Disponible: <https://www.quora.com/How-does-an-event-loop-work>

Lea Eventloop en línea: <https://riptutorial.com/es/node-js/topic/8652/eventloop>

Capítulo 48: Evitar el infierno de devolución de llamada

Examples

Módulo asíncrono

La fuente está disponible para descargar desde GitHub. Alternativamente, puedes instalar usando npm:

```
$ npm instalar --save async
```

Además de usar Bower:

```
$ bower instalar async
```

Ejemplo:

```
var async = require("async");
async.parallel([
  function(callback) { ... },
  function(callback) { ... }
], function(err, results) {
  // optional callback
});
```

Módulo asíncrono

Afortunadamente, existen bibliotecas como Async.js para tratar de frenar el problema. Async agrega una capa delgada de funciones sobre su código, pero puede reducir la complejidad al evitar el anidamiento de devolución de llamada.

Existen muchos métodos de ayuda en Async que se pueden usar en diferentes situaciones, como series, paralelo, cascada, etc. Cada función tiene un caso de uso específico, así que tómese un tiempo para aprender cuál le ayudará en qué situaciones.

Tan bueno como Async es, como todo, no es perfecto. Es muy fácil dejarse llevar por la combinación de series, paralelos, para siempre, etc., momento en el que regresa a donde comenzó con el código desordenado. Tenga cuidado de no optimizar prematuramente. El hecho de que algunas tareas asíncronas se puedan ejecutar en paralelo no siempre significa que deban hacerlo. En realidad, dado que Node es solo un subproceso, la ejecución de tareas en paralelo en el uso de Async tiene poco o ningún aumento de rendimiento.

La fuente está disponible para descargar desde <https://github.com/caolan/async> . Alternativamente, puedes instalar usando npm:

\$ npm instalar --save async

Además de usar Bower:

\$ bower instalar async

Ejemplo de cascada de Async:

```
var fs = require('fs');
var async = require('async');

var myFile = '/tmp/test';

async.waterfall([
  function(callback) {
    fs.readFile(myFile, 'utf8', callback);
  },
  function(txt, callback) {
    txt = txt + '\nAppended something!';
    fs.writeFile(myFile, txt, callback);
  }
], function (err, result) {
  if(err) return console.log(err);
  console.log('Appended text!');
});
```

Lea Evitar el infierno de devolución de llamada en línea: <https://riptutorial.com/es/node-js/topic/10045/evitar-el-infierno-de-devolucion-de-llamada>

Capítulo 49: Exigir()

Introducción

Esta documentación se centra en explicar los usos y la declaración `require()` que [NodeJS](#) incluye en su idioma.

Require es una importación de ciertos archivos o paquetes utilizados con los módulos de NodeJS. Se utiliza para mejorar la estructura del código y los usos. `require()` se usa en archivos que se instalan localmente, con una ruta directa desde el archivo que se `require` .

Sintaxis

- `module.exports = {testFunction: testFunction};`
- `var test_file = require ('./ testFile.js');` // Tengamos un archivo llamado `testFile`
- `test_file.testFunction (our_data);` // Deje que `testFile` tenga la función `testFunction`

Observaciones

El uso de `require()` permite que el código se estructure de manera similar al uso de las [clases](#) y los métodos públicos de Java. Si una función es `.export 'ed`, puede ser `require` en otro archivo para ser utilizada. Si un archivo no es `.export 'ed`, no se puede utilizar en otro archivo.

Examples

A partir del uso `require ()` con una función y archivo.

Requerir es una declaración que Node interpreta como, en cierto sentido, una función `getter` . Por ejemplo, supongamos que tiene un archivo llamado `analysis.js` , y el interior de su archivo se ve así:

```
function analyzeWeather(weather_data) {
  console.log('Weather information for ' + weather_data.time + ': ');
  console.log('Rainfall: ' + weather_data.precip);
  console.log('Temperature: ' + weather_data.temp);
  //More weather_data analysis/printing...
}
```

Este archivo contiene solo el método, `analyzeWeather(weather_data)` . Si queremos usar esta función, se debe usar dentro de este archivo, o se debe copiar en el archivo en el que quiere ser utilizada. Sin embargo, Node ha incluido una herramienta muy útil para ayudar con la organización de códigos y archivos, que son los [módulos](#) .

Para utilizar nuestra función, primero debemos `export` la función a través de una declaración al principio. Nuestro nuevo archivo se ve así,

```

module.exports = {
  analyzeWeather: analyzeWeather
}
function analyzeWeather(weather_data) {
  console.log('Weather information for ' + weather_data.time + ': ');
  console.log('Rainfall: ' + weather_data.precip);
  console.log('Temperature: ' + weather_data.temp);
  //More weather_data analysis/printing...
}

```

Con esta pequeña declaración `module.exports` , nuestra función ahora está lista para usar fuera del archivo. Todo lo que queda por hacer es usar `require()` .

Cuando se `require` una función o un archivo, la sintaxis es muy similar. Por lo general, se realiza al principio del archivo y se establece en `var` 's o `const` 's para su uso en todo el archivo. Por ejemplo, tenemos otro archivo (en el mismo nivel que `analyze.js` llamado `handleWeather.js` que tiene este aspecto,

```

const analysis = require('./analysis.js');

weather_data = {
  time: '01/01/2001',
  precip: 0.75,
  temp: 78,
  //More weather data...
};
analysis.analyzeWeather(weather_data);

```

En este archivo, estamos utilizando `require()` para capturar nuestro archivo `analysis.js` . Cuando se usa, solo llamamos a la variable o constante asignada a este `require` y usamos la función que se exporta dentro.

A partir del uso `require ()` con un paquete NPM

El `require` del nodo también es muy útil cuando se usa en conjunto con un [paquete NPM](#) . Digamos, por ejemplo, que le gustaría usar el paquete NPM `request` en un archivo llamado `getWeather.js` . Después de que [NPM haya instalado](#) su paquete a través de su línea de comando (`git install request`), estará listo para usarlo. Su archivo `getWeather.js` podría gustarle mirar esto,

```

var https = require('request');

//Construct your url variable...
https.get(url, function(error, response, body) {
  if (error) {
    console.log(error);
  } else {
    console.log('Response => ' + response);
    console.log('Body => ' + body);
  }
});

```

Cuando se ejecuta este archivo, primero se `require` (importa) el paquete que acaba de instalar

llamado `request` . Dentro del archivo de `request` , hay muchas funciones a las que ahora tiene acceso, una de las cuales se llama `get` . En las siguientes dos líneas, la función se utiliza para realizar una [solicitud GET de HTTP](#) .

Lea `Exigir()` en línea: <https://riptutorial.com/es/node-js/topic/10742/exigir-->

Capítulo 50: Exportando e importando el módulo en node.js

Examples

Usando un módulo simple en node.js

Qué es un módulo node.js ([enlace al artículo](#)):

Un módulo encapsula código relacionado en una sola unidad de código. Al crear un módulo, esto puede interpretarse como mover todas las funciones relacionadas a un archivo.

Ahora veamos un ejemplo. Imagina que todos los archivos están en el mismo directorio:

Archivo: `printer.js`

```
"use strict";

exports.printHelloWorld = function () {
  console.log("Hello World!!!");
}
```

Otra forma de utilizar módulos:

Archivo `animals.js`

```
"use strict";

module.exports = {
  lion: function() {
    console.log("ROAARR!!!");
  }
};
```

Archivo: `app.js`

Ejecute este archivo yendo a su directorio y escribiendo: `node app.js`

```
"use strict";

//require('./path/to/module.js') node which module to load
var printer = require('./printer');
var animals = require('./animals');

printer.printHelloWorld(); //prints "Hello World!!!"
animals.lion(); //prints "ROAARR!!!"
```

Usando Importaciones En ES6

Node.js está construido contra versiones modernas de V8. Al mantenernos actualizados con las últimas versiones de este motor, nos aseguramos de que las nuevas características de la especificación ECMA-262 de JavaScript se presenten a los desarrolladores de Node.js de manera oportuna, así como mejoras continuas de rendimiento y estabilidad.

Todas las características de ECMAScript 2015 (ES6) se dividen en tres grupos para las características de envío, por etapas y en curso:

Todas las funciones de envío, que V8 considera estables, están activadas de forma predeterminada en Node.js y no requieren ningún tipo de indicador de tiempo de ejecución. Las funciones en etapas, que son funciones casi completas que no son consideradas estables por el equipo V8, requieren una marca de tiempo de ejecución: `--harmony`. Las funciones en curso se pueden activar individualmente por su respectivo indicador de armonía, aunque esto no es recomendable a menos que sea para propósitos de prueba. Nota: estas banderas están expuestas por V8 y potencialmente cambiarán sin ningún aviso de desaprobación.

Actualmente ES6 admite declaraciones de importación de forma nativa. [Consulte aquí](#)

Así que si tenemos un archivo llamado `fun.js` ...

```
export default function say(what){
  console.log(what);
}

export function sayLoud(whoot) {
  say(whoot.toUpperCase());
}
```

... y si hubiera otro archivo llamado `app.js` en el que deseamos poner en uso nuestras funciones previamente definidas, hay tres formas de importarlas.

Importar por defecto

```
import say from './fun';
say('Hello Stack Overflow!!'); // Output: Hello Stack Overflow!!
```

Importa la función `say()` porque está marcada como la exportación predeterminada en el archivo de origen (`export default ...`)

Importaciones con nombre

```
import { sayLoud } from './fun';
sayLoud('JS modules are awesome.');// Output: JS MODULES ARE AWESOME.
```

Las importaciones con nombre nos permiten importar exactamente las partes de un módulo que realmente necesitamos. Hacemos esto nombrándolos explícitamente. En nuestro caso, nombrando a `sayLoud` en paréntesis dentro de la declaración de importación.

Importación agrupada

```
import * as i from './fun';  
i.say('What?'); // Output: What?  
i.sayLoud('Whoot!'); // Output: WHOOT!
```

Si queremos tenerlo todo, este es el camino a seguir. Al utilizar la sintaxis `* as i`, tenemos la declaración de `import` que nos proporciona un objeto `i` que contiene todas las exportaciones de nuestro módulo de `fun` como propiedades con el nombre correspondiente.

Camino

Tenga en cuenta que debe marcar explícitamente sus rutas de importación como rutas *relativas*, incluso si el archivo que desea importar se encuentra en el mismo directorio que el archivo que está importando utilizando `./`. Importaciones desde rutas no prefijadas como

```
import express from 'express';
```

se buscará en las carpetas de `node_modules` locales y globales y arrojará un error si no se encuentran módulos coincidentes.

Exportando con sintaxis ES6

Este es el equivalente del [otro ejemplo](#), pero en su lugar utiliza ES6.

```
export function printHelloWorld() {  
  console.log("Hello World!!!");  
}
```

Lea [Exportando e importando el módulo en node.js en línea: https://riptutorial.com/es/node-js/topic/1173/exportando-e-importando-el-modulo-en-node-js](https://riptutorial.com/es/node-js/topic/1173/exportando-e-importando-el-modulo-en-node-js)

Capítulo 51: Exportando y consumiendo módulos

Observaciones

Mientras que todo en Node.js generalmente se hace de forma asíncrona, `require()` no es una de esas cosas. Como los módulos en la práctica solo necesitan cargarse una vez, es una operación de bloqueo y deben usarse correctamente.

Los módulos se almacenan en caché después de la primera vez que se cargan. Si está editando un módulo en desarrollo, deberá eliminar su entrada en la memoria caché del módulo para poder utilizar nuevos cambios. Dicho esto, incluso si un módulo se borra de la memoria caché del módulo, el módulo en sí no se recolecta como basura, por lo que se debe tener cuidado para su uso en entornos de producción.

Examples

Cargando y utilizando un módulo

Un módulo puede ser "importado", o de lo contrario "requerido" por la función `require()`. Por ejemplo, para cargar el módulo `http` que se envía con Node.js, se puede usar lo siguiente:

```
const http = require('http');
```

Aparte de los módulos que se envían con el tiempo de ejecución, también puede requerir módulos que haya instalado desde npm, como Express. Si ya había instalado Express en su sistema a través de `npm install express`, simplemente podría escribir:

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

También puede incluir módulos que haya escrito usted mismo como parte de su aplicación. En este caso, para incluir un archivo llamado `lib.js` en el mismo directorio que el archivo actual:

```
const mylib = require('./lib');
```

Tenga en cuenta que puede omitir la extensión, y se asumirá `.js`. Una vez que carga un módulo, la variable se llena con un objeto que contiene los métodos y las propiedades publicadas desde el archivo requerido. Un ejemplo completo:

```
const http = require('http');

// The `http` module has the property `STATUS_CODES`
console.log(http.STATUS_CODES[404]); // outputs 'Not Found'
```

```
// Also contains `createServer()`  
http.createServer(function(req, res) {  
  res.writeHead(200, {'Content-Type': 'text/html'});  
  res.write('<html><body>Module Test</body></html>');  
  res.end();  
}).listen(80);
```

Creando un módulo hello-world.js

Node proporciona la interfaz `module.exports` para exponer funciones y variables a otros archivos. La forma más sencilla de hacerlo es exportar solo un objeto (función o variable), como se muestra en el primer ejemplo.

hola-mundo.js

```
module.exports = function(subject) {  
  console.log('Hello ' + subject);  
};
```

Si no queremos que la exportación completa sea un solo objeto, podemos exportar funciones y variables como propiedades del objeto de `exports`. Los tres ejemplos siguientes demuestran esto de maneras ligeramente diferentes:

- `hello-venus.js`: la definición de la función se realiza por separado y luego se agrega como una propiedad de `module.exports`
- `hello-jupiter.js`: las definiciones de funciones se ponen directamente como el valor de las propiedades de `module.exports`
- `hello-mars.js`: la definición de la función se declara directamente como una propiedad de las `exports` que es una versión corta de `module.exports`

hola-venus.js

```
function hello(subject) {  
  console.log('Venus says Hello ' + subject);  
}  
  
module.exports = {  
  hello: hello  
};
```

hola-jupiter.js

```
module.exports = {  
  hello: function(subject) {  
    console.log('Jupiter says hello ' + subject);  
  },  
  
  bye: function(subject) {  
    console.log('Jupiter says goodbye ' + subject);  
  }  
};
```

hola-mars.js

```
exports.hello = function(subject) {  
  console.log('Mars says Hello ' + subject);  
};
```

Módulo de carga con nombre de directorio

Tenemos un directorio llamado `hello` que incluye los siguientes archivos:

index.js

```
// hello/index.js  
module.exports = function(){  
  console.log('Hej');  
};
```

main.js

```
// hello/main.js  
// We can include the other files we've defined by using the `require()` method  
var hw = require('./hello-world.js'),  
    hm = require('./hello-mars.js'),  
    hv = require('./hello-venus.js'),  
    hj = require('./hello-jupiter.js'),  
    hu = require('./index.js');  
  
// Because we assigned our function to the entire `module.exports` object, we  
// can use it directly  
hw('World!'); // outputs "Hello World!"  
  
// In this case, we assigned our function to the `hello` property of exports, so we must  
// use that here too  
hm.hello('Solar System!'); // outputs "Mars says Hello Solar System!"  
  
// The result of assigning module.exports at once is the same as in hello-world.js  
hv.hello('Milky Way!'); // outputs "Venus says Hello Milky Way!"  
  
hj.hello('Universe!'); // outputs "Jupiter says hello Universe!"  
hj.bye('Universe!'); // outputs "Jupiter says goodbye Universe!"  
  
hu(); //output 'hej'
```

Invalidando el caché del módulo

En el desarrollo, es posible que el uso de `require()` en el mismo módulo varias veces siempre devuelva el mismo módulo, incluso si ha realizado cambios en ese archivo. Esto se debe a que los módulos se almacenan en caché la primera vez que se cargan, y cualquier carga subsiguiente del módulo se cargará desde el caché.

Para solucionar este problema, tendrá que `delete` la entrada en el caché. Por ejemplo, si has cargado un módulo:

```
var a = require('./a');
```

A continuación, podría eliminar la entrada de caché:

```
var rpath = require.resolve('./a.js');
delete require.cache[rpath];
```

Y luego volver a requerir el módulo:

```
var a = require('./a');
```

Tenga en cuenta que esto no se recomienda en producción porque la `delete` solo eliminará la referencia al módulo cargado, no los datos cargados en sí. El módulo no se recolecta en la basura, por lo que el uso incorrecto de esta función podría provocar una pérdida de memoria.

Construyendo tus propios módulos.

También puede hacer referencia a un objeto para exportar públicamente y agregar continuamente métodos a ese objeto:

```
const auth = module.exports = {}
const config = require('../config')
const request = require('request')

auth.email = function (data, callback) {
  // Authenticate with an email address
}

auth.facebook = function (data, callback) {
  // Authenticate with a Facebook account
}

auth.twitter = function (data, callback) {
  // Authenticate with a Twitter account
}

auth.slack = function (data, callback) {
  // Authenticate with a Slack account
}

auth.stack_overflow = function (data, callback) {
  // Authenticate with a Stack Overflow account
}
```

Para usar cualquiera de estos, solo necesita el módulo como lo haría normalmente:

```
const auth = require('./auth')

module.exports = function (req, res, next) {
  auth.facebook(req.body, function (err, user) {
    if (err) return next(err)

    req.user = user
    next()
  })
}
```

```
    })  
  }  
}
```

Cada módulo inyectado solo una vez.

NodeJS ejecuta el módulo solo la primera vez que lo requiera. Cualquier otra función requerida volverá a utilizar el mismo Objeto, por lo que no ejecutará el código en el módulo otra vez. Además, Node almacena en caché los módulos la primera vez que se cargan utilizando require. Esto reduce el número de lecturas de archivos y ayuda a acelerar la aplicación.

myModule.js

```
console.log(123) ;  
exports.var1 = 4 ;
```

index.js

```
var a=require('./myModule') ; // Output 123  
var b=require('./myModule') ; // No output  
console.log(a.var1) ; // Output 4  
console.log(b.var1) ; // Output 4  
a.var2 = 5 ;  
console.log(b.var2) ; // Output 5
```

Módulo cargando desde node_modules

Los módulos pueden `require` d sin usar rutas relativas colocándolos en un directorio especial llamado `node_modules` .

Por ejemplo, para `require` un módulo llamado `foo` desde un archivo `index.js` , puede usar la siguiente estructura de directorios:

```
index.js  
  \- node_modules  
    \- foo  
        \- foo.js  
          \- package.json
```

Los módulos se deben colocar dentro de un directorio, junto con un archivo `package.json` . El campo `main` del archivo `package.json` debe apuntar al punto de entrada de su módulo: este es el archivo que se importa cuando los usuarios lo `require('your-module')` . `main` valores predeterminados `main index.js` si no se proporcionan. Como alternativa, puede hacer referencia a los archivos relativos a su módulo, simplemente añadiendo el recorrido relativo al `require` llamada: `require('your-module/path/to/file')` .

Los módulos también pueden `require` desde los directorios de `node_modules` hasta la jerarquía del sistema de archivos. Si tenemos la siguiente estructura de directorios:

```
my-project  
  \- node_modules
```

```
| - foo    // the foo module
| - ...
| - baz    // the baz module
| - node_modules
| - bar    // the bar module
```

podremos `require` el módulo `foo` desde cualquier archivo dentro de la `bar` usando `require('foo')` .

Tenga en cuenta que el nodo solo coincidirá con el módulo más cercano al archivo en la jerarquía del sistema de archivos, comenzando desde (el directorio actual del archivo / `node_modules`). El nodo coincide con los directorios de esta manera hasta la raíz del sistema de archivos.

Puede instalar nuevos módulos desde el registro npm u otros registros npm, o puede crear los suyos propios.

Carpeta como modulo

Los módulos se pueden dividir en muchos archivos `.js` en la misma carpeta. Un ejemplo en una carpeta `my_module` :

`function_one.js`

```
module.exports = function() {
  return 1;
}
```

`function_two.js`

```
module.exports = function() {
  return 2;
}
```

`index.js`

```
exports.f_one = require('./function_one.js');
exports.f_two = require('./function_two.js');
```

Un módulo como este se usa refiriéndose a él por el nombre de la carpeta:

```
var split_module = require('./my_module');
```

Tenga en cuenta que si lo requirió omitiendo `./` o cualquier indicación de una ruta a una carpeta desde el argumento de la función requerida, Node intentará cargar un módulo desde la carpeta `node_modules` .

Alternativamente, puede crear en la misma carpeta un archivo `package.json` con estos contenidos:

```
{
  "name": "my_module",
  "main": "./your_main_entry_point.js"
```

```
}
```

De esta manera, no es necesario que nombre el archivo de módulo principal "índice".

Lea **Exportando y consumiendo módulos en línea**: <https://riptutorial.com/es/node-js/topic/547/exportando-y-consumiendo-modulos>

Capítulo 52: Gestión de errores Node.js

Introducción

Aprenderemos cómo crear objetos de error y cómo lanzar y manejar errores en Node.js

Futuras ediciones relacionadas con las mejores prácticas en el manejo de errores.

Examples

Creando objeto de error

nuevo error (mensaje)

Crea un nuevo objeto de error, donde el `message` valor se establece en propiedad de `message` del objeto creado. Normalmente, los argumentos del `message` se pasan al constructor de errores como una cadena. Sin embargo, si el argumento del `message` es un objeto y no una cadena, el constructor de error llama `.toString()` método `.toString()` del objeto pasado y establece ese valor en la propiedad del `message` del objeto de error creado.

```
var err = new Error("The error message");
console.log(err.message); //prints: The error message
console.log(err);
//output
//Error: The error message
//    at ...
```

Cada objeto de error tiene un seguimiento de pila. El seguimiento de la pila contiene la información del mensaje de error y muestra dónde ocurrió el error (la salida anterior muestra la pila de error). Una vez que se crea el objeto de error, el sistema captura el seguimiento de la pila del error en la línea actual. Para obtener el seguimiento de pila, utilice la propiedad de pila de cualquier objeto de error creado. Debajo de dos líneas son idénticas:

```
console.log(err);
console.log(err.stack);
```

Error de lanzamiento

El error de lanzamiento significa una excepción si no se maneja alguna excepción, el servidor de nodos se bloqueará.

La siguiente línea arroja error:

```
throw new Error("Some error occurred");
```

```
var err = new Error("Some error occurred");
throw err;
```

0

```
throw "Some error occurred";
```

El último ejemplo (lanzar cadenas) no es una buena práctica y no se recomienda (siempre arroje errores que son instancias del objeto Error).

Tenga en cuenta que si `throw` un error en el suyo, entonces el sistema se bloqueará en esa línea (si no hay controladores de excepción), no se ejecutará ningún código después de esa línea.

```
var a = 5;
var err = new Error("Some error message");
throw err; //this will print the error stack and node server will stop
a++; //this line will never be executed
console.log(a); //and this one also
```

Pero en este ejemplo:

```
var a = 5;
var err = new Error("Some error message");
console.log(err); //this will print the error stack
a++;
console.log(a); //this line will be executed and will print 6
```

prueba ... atrapa bloque

`try ... catch` block es para manejar excepciones, recuerde la excepción significa que el error arrojado no el error.

```
try {
  var a = 1;
  b++; //this will cause an error because b is undefined
  console.log(b); //this line will not be executed
} catch (error) {
  console.log(error); //here we handle the error caused in the try block
}
```

En el bloque de `try` `b++` produce un error y ese error se pasa al bloque de `catch` que puede manejarse allí o incluso se puede lanzar el mismo error en el bloque de captura o hacer una pequeña modificación y luego lanzar. Veamos el siguiente ejemplo.

```
try {
  var a = 1;
  b++;
  console.log(b);
} catch (error) {
  error.message = "b variable is undefined, so the undefined can't be incremented"
  throw error;
}
```

En el ejemplo anterior modificamos la propiedad de `message` del objeto de `error` y luego lanzamos el `error` modificado.

Puede a través de cualquier error en su bloque `try` y manejarlo en el bloque `catch`:

```
try {  
  var a = 1;  
  throw new Error("Some error message");  
  console.log(a); //this line will not be executed;  
} catch (error) {  
  console.log(error); //will be the above thrown error  
}
```

Lea Gestión de errores Node.js en línea: <https://riptutorial.com/es/node-js/topic/8590/gestion-de-errores-node-js>

Capítulo 53: Gestor de paquetes de hilo

Introducción

Yarn es un gestor de paquetes para Node.js, similar a npm. Si bien se comparten muchos puntos en común, existen algunas diferencias clave entre Yarn y npm.

Examples

Instalación de hilo

Este ejemplo explica los diferentes métodos para instalar Yarn para su sistema operativo.

Mac OS

Homebrew

```
brew update  
brew install yarn
```

MacPorts

```
sudo port install yarn
```

Agregando Hilo a su RUTA

Agregue lo siguiente a su perfil de shell preferido (`.profile` , `.bashrc` , `.zshrc` etc.)

```
export PATH="$PATH:`yarn global bin`"
```

Windows

Instalador

Primero, instale Node.js si aún no está instalado.

Descargue el instalador de Yarn como `.msi` desde el [sitio web de Yarn](https://yarnpkg.com) .

Chocolatey

```
choco install yarn
```

Linux

Debian / Ubuntu

Asegúrese de que Node.js esté instalado para su distribución, o ejecute lo siguiente

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -  
sudo apt-get install -y nodejs
```

Configurar el repositorio YarnPkg

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -  
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee  
/etc/apt/sources.list.d/yarn.list
```

Instalar hilo

```
sudo apt-get update && sudo apt-get install yarn
```

CentOS / Fedora / RHEL

Instala Node.js si aún no está instalado

```
curl --silent --location https://rpm.nodesource.com/setup_6.x | bash -
```

Instalar hilo

```
sudo wget https://dl.yarnpkg.com/rpm/yarn.repo -O /etc/yum.repos.d/yarn.repo  
sudo yum install yarn
```

Arco

Instale el hilo a través de AUR.

Ejemplo usando yaourt:

```
yaourt -S yarn
```

Solus

```
sudo eoipkg install yarn
```

Todas las distribuciones

Agregue lo siguiente a su perfil de shell preferido (`.profile` , `.bashrc` , `.zshrc` , etc.)

```
export PATH="$PATH:`yarn global bin`"
```

Método alternativo de instalación

Script de shell

```
curl -o- -L https://yarnpkg.com/install.sh | bash
```

o especifique una versión para instalar

```
curl -o- -L https://yarnpkg.com/install.sh | bash -s -- --version [version]
```

Tarball

```
cd /opt
wget https://yarnpkg.com/latest.tar.gz
tar zxvf latest.tar.gz
```

Npm

Si ya tiene npm instalado, simplemente ejecute

```
npm install -g yarn
```

Instalación posterior

Compruebe la versión instalada de Yarn ejecutando

```
yarn --version
```

Creando un paquete básico

El comando `yarn init` lo guiará a través de la creación de un archivo `package.json` para configurar cierta información sobre su paquete. Esto es similar al comando `npm init` en npm.

Cree y navegue a un nuevo directorio para guardar su paquete, y luego ejecute `yarn init`

```
mkdir my-package && cd my-package
yarn init
```

Responde las preguntas que siguen en el CLI.

```
question name (my-package): my-package
question version (1.0.0):
question description: A test package
question entry point (index.js):
question repository url:
question author: StackOverflow Documentation
question license (MIT):
success Saved package.json
[] Done in 27.31s.
```

Esto generará un archivo `package.json` similar al siguiente

```
{
  "name": "my-package",
  "version": "1.0.0",
  "description": "A test package",
  "main": "index.js",
  "author": "StackOverflow Documentation",
  "license": "MIT"
}
```

Ahora intentemos agregar una dependencia. La sintaxis básica para esto es `yarn add [package-name]`

Ejecuta lo siguiente para instalar ExpressJS

```
yarn add express
```

Esto agregará una sección de `dependencies` a su `package.json` , y agregará ExpressJS

```
"dependencies": {
  "express": "^4.15.2"
}
```

Instalar el paquete con hilo

Yarn usa el mismo registro que npm. Eso significa que cada paquete que está disponible en npm es el mismo en Yarn.

Para instalar un paquete, ejecute `yarn add package` .

Si necesita una versión específica del paquete, puede usar `yarn add package@version` .

Si la versión que necesita instalar ha sido etiquetada, puede usar `yarn add package@tag` .

Lea **Gestor de paquetes de hilo en línea**: <https://riptutorial.com/es/node-js/topic/9441/gestor-de-paquetes-de-hilo>

Capítulo 54: gruñido

Observaciones

Otras lecturas:

La [guía](#) de instalación de Grunt tiene información detallada sobre la instalación de versiones específicas de Grunt y grunt-cli, de producción o en desarrollo.

La [guía de configuración de tareas](#) tiene una explicación detallada sobre cómo configurar tareas, objetivos, opciones y archivos dentro de Gruntfile, junto con una explicación de plantillas, patrones globales e importación de datos externos.

La [guía de Creación de Tareas](#) enumera las diferencias entre los tipos de tareas Grunt y muestra una serie de tareas y configuraciones de muestra.

Examples

Introducción a GruntJs

Grunt es un JavaScript Task Runner, que se utiliza para la automatización de tareas repetitivas como minificación, compilación, prueba de unidades, alineación, etc.

Para comenzar, querrá instalar la interfaz de línea de comandos (CLI) de Grunt globalmente.

```
npm install -g grunt-cli
```

Preparación de un nuevo proyecto Grunt: una configuración típica implicará agregar dos archivos a su proyecto: package.json y Gruntfile.

package.json: npm utiliza este archivo para almacenar metadatos para proyectos publicados como módulos npm. Enumera grunt y los complementos de Grunt que su proyecto necesita como DevDependencies en este archivo.

Gruntfile: este archivo se llama Gruntfile.js y se usa para configurar o definir tareas y cargar complementos de Grunt.

Example package.json:

```
{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-nodeunit": "~0.4.1",
    "grunt-contrib-uglify": "~0.5.0"
  }
}
```

```
}
```

Ejemplo de gruntfile:

```
module.exports = function(grunt) {

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
      options: {
        banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
      },
      build: {
        src: 'src/<%= pkg.name %>.js',
        dest: 'build/<%= pkg.name %>.min.js'
      }
    }
  });

  // Load the plugin that provides the "uglify" task.
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // Default task(s).
  grunt.registerTask('default', ['uglify']);

};
```

Instalación de gruntplugins

Añadiendo dependencia

Para usar un gruntplugin, primero debe agregarlo como una dependencia a su proyecto. Vamos a usar el plugin jshint como ejemplo.

```
npm install grunt-contrib-jshint --save-dev
```

La opción `--save-dev` se usa para agregar el complemento en el `package.json`, de esta manera el complemento siempre se instala después de una `npm install`.

Cargando el plugin

Puede cargar su complemento en el archivo `loadNpmTasks` usando `loadNpmTasks`.

```
grunt.loadNpmTasks('grunt-contrib-jshint');
```

Configurando la tarea

Configura la tarea en el archivo `grunt` agregando una propiedad llamada `jshint` al objeto pasado a `grunt.initConfig`.

```
grunt.initConfig({
  jshint: {
```

```
    all: ['Gruntfile.js', 'lib/**/*.js', 'test/**/*.js']
  }
});
```

No olvide que puede tener otras propiedades para otros complementos que está utilizando.

Ejecutando la tarea

Para ejecutar la tarea con el complemento, puede utilizar la línea de comandos.

```
grunt jshint
```

O puede agregar `jshint` a otra tarea.

```
grunt.registerTask('default', ['jshint']);
```

La tarea predeterminada se ejecuta con el comando `grunt` en el terminal sin ninguna opción.

Lea grunido en línea: <https://riptutorial.com/es/node-js/topic/6059/grunido>

Capítulo 55: Guía para principiantes de NodeJS

Examples

Hola Mundo !

Coloque el siguiente código en un nombre de archivo `helloworld.js`

```
console.log("Hello World");
```

Guarde el archivo y ejecútelo a través de Node.js:

```
node helloworld.js
```

Lea Guía para principiantes de NodeJS en línea: <https://riptutorial.com/es/node-js/topic/7693/guia-para-principiantes-de-nodejs>

Capítulo 56: herrero

Examples

Construye un blog simple

Suponiendo que tiene nodos y npm instalados y disponibles, cree una carpeta de proyecto con un `package.json` válido. Instalar las dependencias necesarias:

```
npm install --save-dev metalsmith metalsmith-in-place handlebars
```

Cree un archivo llamado `build.js` en la raíz de su carpeta de proyecto, que contenga lo siguiente:

```
var metalsmith = require('metalsmith');
var handlebars = require('handlebars');
var inPlace = require('metalsmith-in-place');

Metalsmith(__dirname)
  .use(inPlace('handlebars'))
  .build(function(err) {
    if (err) throw err;
    console.log('Build finished!');
  });
```

Cree una carpeta llamada `src` en la raíz de la carpeta de su proyecto. Cree `index.html` en `src`, que contiene lo siguiente:

```
---
title: My awesome blog
---
<h1>{{ title }}</h1>
```

La ejecución del `node build.js` ahora construirá todos los archivos en `src`. Después de ejecutar este comando, tendrás `index.html` en tu carpeta de compilación, con los siguientes contenidos:

```
<h1>My awesome blog</h1>
```

Lea herrero en línea: <https://riptutorial.com/es/node-js/topic/6111/herrero>

Capítulo 57: Historia de Nodejs

Introducción

Aquí vamos a discutir sobre la historia de Node.js, la información de la versión y su estado actual.

Examples

Eventos clave en cada año.

2009

- 3 de marzo: [el proyecto fue nombrado como "nodo"](#)
- 1 de octubre: [primera vista previa muy temprana de npm, el paquete Node gerente](#)
- 8 de noviembre: [Node.js original de Ryan Dahl \(Creador de Node.js\) en JSConf 2009](#)

2010

- Express: Un marco de desarrollo web Node.js
- Socket.io lanzamiento inicial
- 28 de abril: [Soporte de Experimental Node.js en Heroku](#)
- 28 de julio: [Google Tech Talk de Ryan Dahl en Node.js](#)
- 20 de agosto: [Node.js 0.2.0 liberado.](#)

2011

- 31 de marzo: Guía Node.js
- 1 de mayo: [npm 1.0: estrenada](#)
- 1 de mayo: [AMA de Ryan Dahl en Reddit](#)
- 10 de julio: [Se ha completado el Libro para principiantes de Node, una introducción a Node.js.](#)
 - Un completo tutorial de Node.js para principiantes.
- 16 de agosto: [LinkedIn usa Node.js](#)
 - LinkedIn lanzó su aplicación móvil completamente renovada con nuevas características y nuevas partes bajo el capó.
- 5 de octubre: [Ryan Dahl habla sobre la historia de Node.js y por qué la creó.](#)
- 5 de diciembre: [Node.js en producción en Uber.](#)
 - El Gerente de Ingeniería de Uber, Curtis Chambers, explica por qué su empresa rediseñó completamente su aplicación utilizando Node.js para aumentar la eficiencia y mejorar la experiencia del socio y del cliente.

2012

- 30 de enero: el [creador de Node.js](#), Ryan Dahl, se aleja del día a día de Node
- 25 de junio: [Node.js v0.8.0 \[stable\]](#) está fuera
- 20 de diciembre: se lanza [Hapi](#), un [framework Node.js](#).

2013

- 30 de abril: [The MEAN Stack: MongoDB, ExpressJS, AngularJS y Node.js](#)
- 17 de mayo: [Cómo construimos la primera aplicación de eBay Node.js](#)
- 15 de noviembre: [PayPal lanza Kraken](#), un [framework Node.js](#)
- 22 de noviembre: [Node.js Memory Fuga en Walmart](#)
 - Eran Hammer de los laboratorios de Wal-Mart llegó al equipo central de Node.js quejándose de una pérdida de memoria que había estado rastreando durante meses.
- 19 de diciembre: [Koa](#) - Framework web para Node.js

2014

- 15 de enero: [TJ Fontaine asume el proyecto Node](#).
- 23 de octubre: [Node.js Advisory Board](#)
 - Joyent y varios miembros de la comunidad Node.js anunciaron una propuesta para una Junta Asesora de Node.js como el siguiente paso hacia un modelo de gobierno completamente abierto para el proyecto de código abierto Node.js.
- 19 de noviembre: [Node.js en Flame Graphs - Netflix](#)
- 28 de noviembre: [IO.js](#) - E / S con evento para V8 Javascript

2015

Q1

- 14 de enero: [IO.js 1.0.0](#)
- Décimo siglo: [Joyent se mueve para establecer la Fundación Node.js](#)
 - Joyent, IBM, Microsoft, PayPal, Fidelity, SAP y The Linux Foundation unen fuerzas para apoyar a la comunidad Node.js con gobierno neutral y abierto
- [Propuesta de conciliación de 27th Febraury](#) : [IO.js](#) y [Node.js](#)

Q2

- 14 de abril: [npm Módulos privados](#).
- 28 de mayo: el [líder del nodo TJ Fontaine se retira y abandona a Joyent](#)
- 13 de mayo: [Node.js y io.js se fusionan bajo la Fundación Node](#)

Q3

- 2 de agosto: [seguimiento - monitoreo y depuración del rendimiento de Node.js](#)
 - Trace es una herramienta de monitoreo de microservicio visualizado que le brinda todas las métricas que necesita cuando opera microservicios.
- 13 de agosto: [4.0 es el nuevo 1.0.](#)

Q4

- 12 de octubre: [Nodo v4.2.0, primer lanzamiento de soporte a largo plazo](#)
- 8 de diciembre: [Apigee, RisingStack y Yahoo se unen a la Fundación Node.js](#)
- 8 y 9 de diciembre: [Nodo interactivo.](#)
 - La primera conferencia anual de Node.js por la Fundación Node.js

2016

Q1

- 10 de febrero: [Express se convierte en un proyecto incubado.](#)
- 23 de marzo: [El incidente del leftpad.](#)
- 29 de marzo: [Google Cloud Platform se une a la Fundación Node.js](#)

Q2

- 26 de abril: [npm tiene 210.000 usuarios.](#)

Q3

- 18 de julio: [CJ Silverio se convierte en el CTO de npm.](#)
- 1 de agosto: [Trace, la solución de depuración Node.js está disponible en general.](#)
- 15 de septiembre: [El primer nodo interactivo en Europa.](#)

Q4

- 11 de octubre: [el administrador del paquete de hilados fue liberado.](#)
- 18 de octubre: [Node.js 6 se convierte en la versión LTS.](#)

Referencia

1. "Historia de Node.js en una línea de tiempo" [en línea]. Disponible: [<https://blog.risingstack.com/history-of-node-js>]

Lea Historia de Nodejs en línea: <https://riptutorial.com/es/node-js/topic/8653/historia-de-nodejs>

Capítulo 58: http

Examples

servidor http

Un ejemplo básico de servidor HTTP.

escriba el siguiente código en el archivo http_server.js:

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

var httpPort = 80;

http.createServer(handler).listen(httpPort, start_callback);

function handler(req, res) {

    var clientIP = req.connection.remoteAddress;
    var connectUsing = req.connection.encrypted ? 'SSL' : 'HTTP';
    console.log('Request received: ' + connectUsing + ' ' + req.method + ' ' + req.url);
    console.log('Client IP: ' + clientIP);

    res.writeHead(200, "OK", {'Content-Type': 'text/plain'});
    res.write("OK");
    res.end();
    return;
}

function start_callback(){
    console.log('Start HTTP on port ' + httpPort)
}
```

luego desde su ubicación http_server.js ejecute este comando:

```
node http_server.js
```

Deberías ver este resultado:

```
> Start HTTP on port 80
```

ahora necesita probar su servidor, necesita abrir su navegador de internet y navegar a este url:

```
http://127.0.0.1:80
```

Si su máquina está ejecutando un servidor Linux, puede probarla así:

```
curl 127.0.0.1:80
```

Deberías ver el siguiente resultado:

```
ok
```

En su consola, que al ejecutar la aplicación, verá estos resultados:

```
> Request received: HTTP GET /  
> Client IP: ::ffff:127.0.0.1
```

cliente http

Un ejemplo básico para el cliente http:

escriba el siguiente código en el archivo `http_client.js`:

```
var http = require('http');  
  
var options = {  
  hostname: '127.0.0.1',  
  port: 80,  
  path: '/',  
  method: 'GET'  
};  
  
var req = http.request(options, function(res) {  
  console.log('STATUS: ' + res.statusCode);  
  console.log('HEADERS: ' + JSON.stringify(res.headers));  
  res.setEncoding('utf8');  
  res.on('data', function (chunk) {  
    console.log('Response: ' + chunk);  
  });  
  res.on('end', function (chunk) {  
    console.log('Response ENDED');  
  });  
});  
  
req.on('error', function(e) {  
  console.log('problem with request: ' + e.message);  
});  
  
req.end();
```

luego desde su ubicación `http_client.js` ejecute este comando:

```
node http_client.js
```

Deberías ver este resultado:

```
> STATUS: 200  
> HEADERS: {"content-type":"text/plain","date":"Thu, 21 Jul 2016 11:27:17  
GMT","connection":"close","transfer-encoding":"chunked"}  
> Response: OK  
> Response ENDED
```

nota: este ejemplo depende del ejemplo del servidor http.

Lea http en línea: <https://riptutorial.com/es/node-js/topic/2973/http>

Capítulo 59: Instalación de Node.js

Examples

Instala Node.js en Ubuntu

Usando el gestor de paquetes apt

```
sudo apt-get update
sudo apt-get install nodejs
sudo apt-get install npm
sudo ln -s /usr/bin/nodejs /usr/bin/node

# the node & npm versions in apt are outdated. This is how you can update them:
sudo npm install -g npm
sudo npm install -g n
sudo n stable # (or lts, or a specific version)
```

Usando la última versión específica (ej. LTS 6.x) directamente desde nodesource

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
apt-get install -y nodejs
```

Además, para la forma correcta de instalar módulos npm globales, establezca el directorio personal para ellos (elimina la necesidad de sudo y evita los errores de EACCES):

```
mkdir ~/.npm-global
echo "export PATH=~/.npm-global/bin:$PATH" >> ~/.profile
source ~/.profile
npm config set prefix '~/.npm-global'
```

Instalación de Node.js en Windows

Instalación estándar

Todos los binarios, instaladores y archivos fuente de Node.js se pueden descargar [aquí](#) .

Puede descargar solo el tiempo de ejecución de `node.exe` o usar el instalador de Windows (`.msi`), que también instalará `npm` , el administrador de paquetes recomendado para Node.js y configure las rutas.

Instalación por gestor de paquetes

También puede realizar la instalación mediante el administrador de paquetes [Chocolatey](#) (automatización de administración de software).

```
# choco install nodejs.install
```

Más información sobre la versión actual, puede encontrarla en el repositorio de choco [aquí](#) .

Usando el administrador de versiones de nodos (nvm)

[Node Version Manager](#) , también conocido como nvm, es un script de bash que simplifica la administración de múltiples versiones de Node.js.

Para instalar nvm, use el script de instalación provisto:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Para Windows hay un paquete nvm-windows con un instalador. Esta página de [Github](#) tiene los detalles para instalar y usar el paquete nvm-windows.

Después de instalar nvm, ejecute "nvm on" desde la línea de comandos. Esto permite que nvm controle las versiones del nodo.

Nota: Es posible que deba reiniciar su terminal para que reconozca el comando `nvm` recién instalado.

Luego instale la última versión del nodo:

```
$ nvm install node
```

También puede instalar una versión de Nodo específica, pasando las versiones principal, secundaria y / o parche:

```
$ nvm install 6  
$ nvm install 4.2
```

Para listar las versiones disponibles para instalar:

```
$ nvm ls-remote
```

Luego puede cambiar las versiones pasando la versión de la misma manera que lo hace al instalar:

```
$ nvm use 5
```

Puede configurar una versión específica del nodo que instaló para que sea la **versión predeterminada** ingresando:

```
$ nvm alias default 4.2
```

Para mostrar una lista de las versiones de nodo que están instaladas en su máquina, ingrese:

```
$ nvm ls
```

Para usar versiones de nodo específicas del proyecto, puede guardar la versión en el archivo `.nvmrc`. De esta manera, comenzar a trabajar con otro proyecto será menos propenso a errores después de obtenerlo de su repositorio.

```
$ echo "4.2" > .nvmrc
$ nvm use
Found '/path/to/project/.nvmrc' with version <4.2>
Now using node v4.2 (npm v3.7.3)
```

Cuando Node se instala a través de nvm, no tenemos que usar `sudo` para instalar paquetes globales, ya que están instalados en la carpeta de inicio. Por `npm i -g http-server` tanto, `npm i -g http-server` funciona sin ningún error de permiso.

Instale Node.js From Source con el administrador de paquetes APT

Prerrequisitos

```
sudo apt-get install build-essential
sudo apt-get install python

[optional]
sudo apt-get install git
```

Obtener fuente y construir

```
cd ~
git clone https://github.com/nodejs/node.git
```

O Para la última versión 6.10.2 de LTS Node.js

```
cd ~
wget https://nodejs.org/dist/v6.3.0/node-v6.10.2.tar.gz
tar -xzf node-v6.10.2.tar.gz
```

Cambie al directorio de origen, como en `cd ~/node-v6.10.2`

```
./configure
make
sudo make install
```

Instalando Node.js en Mac usando el administrador de paquetes

Homebrew

Puedes instalar Node.js usando el administrador de paquetes de [Homebrew](#) .

Comience por actualizar brew:

```
brew update
```

Es posible que necesite cambiar permisos o rutas. Es mejor ejecutar esto antes de continuar:

```
brew doctor
```

A continuación, puede instalar Node.js ejecutando:

```
brew install node
```

Una vez que Node.js está instalado, puede validar la versión instalada ejecutando:

```
node -v
```

Macports

También puede instalar node.js a través de [Macports](#) .

Primero actualícelo para asegurarse de que se haga referencia a los paquetes más recientes:

```
sudo port selfupdate
```

Luego instale nodejs y npm

```
sudo port install nodejs npm
```

Ahora puede ejecutar el nodo a través de la CLI directamente invocando el `node` . Además, puede comprobar su versión de nodo actual con

```
node -v
```

Instalación utilizando el instalador de MacOS X

Puede encontrar los instaladores en la [página de descarga de Node.js](#). Normalmente, Node.js recomienda dos versiones de Node, la versión LTS (soporte a largo plazo) y la versión actual (última versión). Si eres nuevo en Node, solo ve al LTS y luego haz clic en el botón del `Macintosh Installer` para descargar el paquete.

Si desea encontrar otras versiones de NodeJS, vaya [aquí](#) , elija su versión y luego haga clic en descargar. Desde la página de descarga, busque un archivo con la extensión `.pkg` .

Una vez que haya descargado el archivo (con la extensión `.pkg` curso), haga doble clic en él para instalarlo. El instalador empaquetado con `Node.js` y `npm` , de forma predeterminada, el paquete instalará ambos, pero puede personalizar cuál instalar haciendo clic en el botón `customize` en el paso `Installation Type` . Aparte de eso, solo sigue las instrucciones de instalación, es bastante sencillo.

Compruebe si Node está instalado

terminal abierta (si no sabe cómo abrir su terminal, mire este [wikihow](#)). En el `node --version` tipo terminal `node --version` luego ingrese. Su terminal se verá así si Node está instalado:

```
$ node --version
v7.2.1
```

La `v7.2.1` es su versión de Node.js, si recibe el `command not found: node` mensaje `command not found: node` lugar de eso, entonces significa que hay un problema con su instalación.

Instalando Node.js en Raspberry PI

Para instalar `v6.x` actualiza los paquetes.

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
```

Usando el gestor de paquetes `apt`

```
sudo apt-get install -y nodejs
```

Instalación con Node Version Manager bajo Fish Shell con Oh My Fish!

[Node Version Manager](#) (`nvm`) simplifica enormemente la administración de las versiones de Node.js, su instalación, y elimina la necesidad de `sudo` cuando se trata de paquetes (por ejemplo, `npm install ...`). [Fish Shell](#) (`fish`) " *es un shell de línea de comandos inteligente y fácil de usar para OS X, Linux y el resto de la familia* ", que es una alternativa popular entre los programadores para shells comunes como `bash` . Por último, [Oh My Fish](#) (`omf`) permite personalizar e instalar paquetes dentro del shell de Fish.

Esta guía asume que ya estás usando Fish como tu caparazón .

Instalar nvm

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash
```

Instale Oh My Fish

```
curl -L https://github.com/oh-my-fish/oh-my-fish/raw/master/bin/install | fish
```

(Nota: Se le solicitará que reinicie su terminal en este punto. Continúe y hágalo ahora).

Instalar plugin-nvm para Oh My Fish

Instalaremos [plugin-nvm](#) a través de Oh My Fish para exponer las capacidades de `nvm` dentro del shell de Fish:

```
omf install nvm
```

Instale Node.js con Node Version Manager

Ahora está listo para usar `nvm`. Puede instalar y usar la versión de Node.js de su agrado. Algunos ejemplos:

- Instale la versión más reciente del nodo: `nvm install node`
- Instale 6.3.1 específicamente: `nvm install 6.3.1`
- Lista de versiones instaladas: `nvm ls`
- Cambie a un 4.3.1 previamente instalado: `nvm use 4.3.1`

Notas finales

¡Recuerda de nuevo, que ya no necesitamos `sudo` cuando tratamos con Node.js usando este método! Las versiones de nodo, los paquetes, etc. se instalan en su directorio de inicio.

Instale Node.js desde la fuente en Centos, RHEL y Fedora

Prerrequisitos

- `git`
- `clang` and `clang++ 3.4` ^ o `gcc` and `g++ 4.8` ^
- Python 2.6 o 2.7
- GNU Make 3.81 ^

Obtener fuente

Node.js v6.x LTS

```
git clone -b v6.x https://github.com/nodejs/node.git
```

Node.js v7.x

```
git clone -b v7.x https://github.com/nodejs/node.git
```

Construir

```
cd node
./configure
make -jX
su -c make install
```

X - el número de núcleos de procesador, acelera enormemente la construcción

Limpieza [Opcional]

```
cd
rm -rf node
```

Instalando Node.js con n

Primero, hay un envoltorio muy bueno para configurar `n` en su sistema. Solo corre:

```
curl -L https://git.io/n-install | bash
```

instalar `n`. Luego instale los binarios de varias maneras:

último

```
n latest
```

estable

```
n stable
```

lts

```
n lts
```

Cualquier otra version

```
n <version>
```

por ejemplo, `n 4.4.7`

Si esta versión ya está instalada, este comando activará esa versión.

Versiones de conmutación

`n` por sí mismo producirá una lista de selección de binarios instalados. Use hacia arriba y hacia abajo para encontrar el que desea y Enter para activarlo.

Lea Instalación de Node.js en línea: <https://riptutorial.com/es/node-js/topic/1294/instalacion-de-node-js>

Capítulo 60: Integracion de cassandra

Examples

Hola Mundo

Para acceder `cassandra-driver` módulo Cassandra `cassandra-driver` desde DataStax se puede usar. Es compatible con todas las características y se puede configurar fácilmente.

```
const cassandra = require("cassandra-driver");
const clientOptions = {
  contactPoints: ["host1", "host2"],
  keyspace: "test"
};

const client = new cassandra.Client(clientOptions);

const query = "SELECT hello FROM world WHERE name = ?";
client.execute(query, ["John"], (err, results) => {
  if (err) {
    return console.error(err);
  }

  console.log(results.rows);
});
```

Lea Integracion de cassandra en línea: <https://riptutorial.com/es/node-js/topic/5949/integracion-de-cassandra>

Capítulo 61: Integración de mongodb

Sintaxis

- `db colección .insertOne ( documento , opciones (w, wtimeout, j, serializeFuntions, forceServerObjectId, bypassDocumentValidation) , devolución de llamada )`
- `db colección .insertMany ( [documentos] , opciones (w, wtimeout, j, serializeFuntions, forceServerObjectId, bypassDocumentValidation) , devolución de llamada )`
- `db colección .find ( consulta )`
- `db colección .updateOne ( filtro , actualización , opciones (upsert, w, wtimeout, j, bypassDocumentValidation) , devolución de llamada )`
- `db colección .updateMany ( filtro , actualización , opciones (upsert, w, wtimeout, j) , devolución de llamada )`
- `db colección .deleteOne ( filtro , opciones (upsert, w, wtimeout, j) , devolución de llamada )`
- `db colección .deleteMany ( filtro , opciones (upsert, w, wtimeout, j) , devolución de llamada )`

Parámetros

Parámetro	Detalles
documento	Un objeto javascript que representa un documento.
documentos	Una serie de documentos
consulta	Un objeto que define una consulta de búsqueda.
filtrar	Un objeto que define una consulta de búsqueda.
llamar de vuelta	Función a llamar cuando se realiza la operación.
opciones	(Opcional) Configuraciones opcionales (por defecto: nulo)
w	(Opcional) La preocupación de escritura
tiempo fuera	(Opcional) El tiempo de espera de escritura escribe. (por defecto: nulo)
j	(opcional) Especifique una preocupación de escritura de diario (predeterminado: falso)
sobresalir	(Opcional) Operación de actualización (por defecto: falso)
multi	(opcional) Actualizar uno / todos los documentos (por defecto: falso)
serializeFunctions	(Opcional) Serializar funciones en cualquier objeto (por defecto:

Parámetro	Detalles
	<i>falso</i>)
forceServerObjectId	(<i>opcional</i>) Forzar al servidor a asignar valores <code>_id</code> en lugar de controlador (<i>predeterminado: falso</i>)
bypassDocumentValidation	(<i>opcional</i>) Permitir que el controlador omita la validación del esquema en MongoDB 3.2 o superior (<i>predeterminado: falso</i>)

Examples

Conectarse a MongoDB

Conéctese a MongoDB, imprima '¡Conectado!' y cierra la conexión.

```
const MongoClient = require('mongodb').MongoClient;

var url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function(err, db) { // MongoClient method 'connect'
  if (err) throw new Error(err);
  console.log("Connected!");
  db.close(); // Don't forget to close the connection when you are done
});
```

Método MongoClient `connect()`

`MongoClient.connect ( url , opciones , devolución de llamada )`

Argumento	Tipo	Descripción
url	cuerda	Una cadena que especifica el servidor IP / nombre de host, puerto y base de datos
options	objeto	(<i>Opcional</i>) Configuraciones opcionales (<i>por defecto: nulo</i>)
callback	Función	Función a llamar cuando se realiza el intento de conexión

La función de `callback` toma dos argumentos

- `err` : Error: si se produce un error, se definirá el argumento `err`
- `db` : object - La instancia de MongoDB

Inserte un documento

Inserte un documento llamado 'myFirstDocument' y establezca 2 propiedades, `greetings` y `farewell`

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').insertOne({ // Insert method 'insertOne'
    "myFirstDocument": {
      "greetings": "Hellu",
      "farewell": "Bye"
    }
  }, function (err, result) {
    if (err) throw new Error(err);
    console.log("Inserted a document into the myCollection collection!");
    db.close(); // Don't forget to close the connection when you are done
  });
});
```

Método de recogida `insertOne()`

`db.collection ( colección ) .insertOne ( documento , opciones , devolución de llamada )`

Argumento	Tipo	Descripción
collection	cuerda	Una cadena que especifica la colección.
document	objeto	El documento a insertar en la colección.
options	objeto	<i>(Opcional)</i> Configuraciones opcionales <i>(por defecto: nulo)</i>
callback	Función	Función a llamar cuando se realiza la operación de inserción

La función de `callback` toma dos argumentos

- `err` : Error: si se produce un error, se definirá el argumento `err`
- `result` : objeto - Un objeto que contiene detalles sobre la operación de inserción

Leer una coleccion

Obtenga todos los documentos de la colección 'myCollection' e imprímalos en la consola.

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  var cursor = db.collection('myCollection').find(); // Read method 'find'
  cursor.each(function (err, doc) {
    if (err) throw new Error(err);
    if (doc != null) {
      console.log(doc); // Print all documents
    } else {

```

```
    db.close(); // Don't forget to close the connection when you are done
  }
});
});
```

Método de recogida `find()`

`db.collection ( colección ) .find ()`

Argumento	Tipo	Descripción
<code>collection</code>	cuerda	Una cadena que especifica la colección.

Actualizar un documento

Encuentre un documento con la propiedad `{ greetings: 'Hellu' }` y cámbielo a `{ greetings: 'Whut?' }`

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').updateOne({ // Update method 'updateOne'
    greetings: "Hellu" },
    { $set: { greetings: "Whut?" } },
    function (err, result) {
      if (err) throw new Error(err);
      db.close(); // Don't forget to close the connection when you are done
    });
});
```

Método de `updateOne()`

`db.collection ( collection ) .updateOne ( filter , update , options . callback )`

Parámetro	Tipo	Descripción
<code>filter</code>	objeto	Especifica la selección crítica.
<code>update</code>	objeto	Especifica las modificaciones a aplicar.
<code>options</code>	objeto	(Opcional) Configuraciones opcionales (por defecto: nulo)
<code>callback</code>	Función	Función a llamar cuando se realiza la operación.

La función de `callback` toma dos argumentos

- `err` : Error: si se produce un error, se definirá el argumento `err`
- `db` : object - La instancia de MongoDB

Borrar un documento

Eliminar un documento con la propiedad `{ greetings: 'Whut?' }`

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').deleteOne(// Delete method 'deleteOne'
    { greetings: "Whut?" },
    function (err, result) {
      if (err) throw new Error(err);
      db.close(); // Don't forget to close the connection when you are done
    });
});
```

Método de `deleteOne()`

`db.collection ( colección ) .deleteOne ( filtro , opciones , devolución de llamada )`

Parámetro	Tipo	Descripción
<code>filter</code>	objeto	Un documento que especifica la selección crítica.
<code>options</code>	objeto	(Opcional) Configuraciones opcionales (por defecto: nulo)
<code>callback</code>	Función	Función a llamar cuando se realiza la operación.

La función de `callback` toma dos argumentos

- `err` : Error: si se produce un error, se definirá el argumento `err`
- `db` : object - La instancia de MongoDB

Eliminar múltiples documentos

Elimine TODOS los documentos con una propiedad de "despedida" establecida en "bien".

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').deleteMany(// MongoDB delete method 'deleteMany'
    { farewell: "okay" }, // Delete ALL documents with the property 'farewell: okay'
    function (err, result) {
```

```

        if (err) throw new Error(err);
        db.close(); // Don't forget to close the connection when you are done
    });
});

```

Método de `deleteMany()`

`db.collection ( colección ) .deleteMany ( filtro , opciones , devolución de llamada )`

Parámetro	Tipo	Descripción
<code>filter</code>	documento	Un documento que especifica la selección crítica.
<code>options</code>	objeto	(Opcional) Configuraciones opcionales (por defecto: nulo)
<code>callback</code>	función	Función a llamar cuando se realiza la operación.

La función de `callback` toma dos argumentos

- `err` : Error: si se produce un error, se definirá el argumento `err`
- `db` : object - La instancia de MongoDB

Conexión simple

```

MongoDB.connect('mongodb://localhost:27017/databaseName', function(error, database) {
    if(error) return console.log(error);
    const collection = database.collection('collectionName');
    collection.insert({key: 'value'}, function(error, result) {
        console.log(error, result);
    });
});

```

Conexión simple, utilizando promesas.

```

const MongoDB = require('mongodb');

MongoDB.connect('mongodb://localhost:27017/databaseName')
    .then(function(database) {
        const collection = database.collection('collectionName');
        return collection.insert({key: 'value'});
    })
    .then(function(result) {
        console.log(result);
    });

```

Lea Integración de mongodb en línea: <https://riptutorial.com/es/node-js/topic/5002/integracion-de-mongodb>

Capítulo 62: Integración de MongoDB para Node.js / Express.js

Introducción

MongoDB es una de las bases de datos NoSQL más populares, gracias a la ayuda de la pila MEAN. La interfaz con una base de datos Mongo desde una aplicación Express es rápida y fácil, una vez que entienda la sintaxis de consulta un tanto torpe. Usaremos Mangosta para ayudarnos.

Observaciones

Puede encontrar más información aquí: <http://mongoosejs.com/docs/guide.html>

Examples

Instalación de MongoDB

```
npm install --save mongodb
npm install --save mongoose //A simple wrapper for ease of development
```

En su archivo de servidor (normalmente denominado index.js o server.js)

```
const express = require('express');
const mongodb = require('mongodb');
const mongoose = require('mongoose');
const mongoConnectionString = 'http://localhost/database name';

mongoose.connect(mongoConnectionString, (err) => {
  if (err) {
    console.log('Could not connect to the database');
  }
});
```

Creando un Modelo de Mangosta

```
const Schema = mongoose.Schema;
const ObjectId = Schema.Types.ObjectId;

const Article = new Schema({
  title: {
    type: String,
    unique: true,
    required: [true, 'Article must have title']
  },
  author: {
    type: ObjectId,
    ref: 'User'
  }
});
```

```

    }
  });

module.exports = mongoose.model('Article', Article);

```

Vamos a analizar esto. MongoDB y Mongoose usan JSON (en realidad BSON, pero eso es irrelevante aquí) como formato de datos. En la parte superior, he establecido algunas variables para reducir la escritura.

Creo un `new Schema` y lo asigno a una constante. Es simple JSON, y cada atributo es otro Objeto con propiedades que ayudan a imponer un esquema más consistente. `Unique` obliga a que se inserten nuevas instancias en la base de datos para, obviamente, ser únicas. Esto es excelente para evitar que un usuario cree varias cuentas en un servicio.

`Requerido` es otro, declarado como una matriz. El primer elemento es el valor booleano y el segundo es el mensaje de error si el valor que se inserta o actualiza no existe.

Los `ObjectId`s se utilizan para relaciones entre modelos. Los ejemplos pueden ser 'Los usuarios tienen muchos comentarios'. Se pueden usar otros atributos en lugar de `ObjectId`. Cadenas como un nombre de usuario es un ejemplo.

Por último, la exportación del modelo para usar con las rutas de su API le brinda acceso a su esquema.

Consultar tu base de datos Mongo

Una simple solicitud GET. Supongamos que el modelo del ejemplo anterior está en el archivo

`./db/models/Article.js`.

```

const express = require('express');
const Articles = require('./db/models/Article');

module.exports = function (app) {
  const routes = express.Router();

  routes.get('/articles', (req, res) => {
    Articles.find().limit(5).lean().exec((err, doc) => {
      if (doc.length > 0) {
        res.send({ data: doc });
      } else {
        res.send({ success: false, message: 'No documents retrieved' });
      }
    });
  });

  app.use('/api', routes);
};

```

Ahora podemos obtener los datos de nuestra base de datos enviando una solicitud HTTP a este punto final. Algunas cosas clave, sin embargo:

1. El límite hace exactamente lo que parece. Solo estoy recibiendo 5 documentos de vuelta.

2. Lean elimina algunas cosas del BSON en bruto, reduciendo la complejidad y los gastos generales. No requerido. Pero útil.
3. Cuando use `find` lugar de `findOne` , confirme que `doc.length` es mayor que 0. Esto se debe a que `find` siempre devuelve una matriz, por lo que una matriz vacía no manejará su error a menos que se verifique la longitud
4. Personalmente me gusta enviar el mensaje de error en ese formato. Cámbiala según tus necesidades. Lo mismo para el documento devuelto.
5. El código en este ejemplo se escribe bajo el supuesto de que lo ha colocado en otro archivo y no directamente en el servidor Express. Para llamar a esto en el servidor, incluya estas líneas en su código de servidor:

```
const app = express();  
require('./path/to/this/file')(app) //
```

Lea Integración de MongoDB para Node.js / Express.js en línea: <https://riptutorial.com/es/node-js/topic/9020/integracion-de-mongodb-para-node-js---express-js>

Capítulo 63: Integración de MySQL

Introducción

En este tema, aprenderá cómo integrarse con Node.js usando la herramienta de administración de bases de datos MYSQL. Aprenderá varias formas de conectarse e interactuar con los datos que residen en mysql mediante un programa y script de nodejs.

Examples

Consultar un objeto de conexión con parámetros.

Cuando desee utilizar el contenido generado por el usuario en el SQL, éste se realiza con los parámetros. Por ejemplo, para buscar usuarios con el nombre `aminadav`, debe hacer:

```
var username = 'aminadav';
var querystring = 'SELECT name, email from users where name = ?';
connection.query(querystring, [username], function(err, rows, fields) {
  if (err) throw err;
  if (rows.length) {
    rows.forEach(function(row) {
      console.log(row.name, 'email address is', row.email);
    });
  } else {
    console.log('There were no results.');
```

Usando un conjunto de conexiones

a. Ejecutando múltiples consultas al mismo tiempo

Todas las consultas en la conexión de MySQL se realizan una tras otra. Esto significa que si desea hacer 10 consultas y cada consulta tarda 2 segundos, se tardará 20 segundos en completar toda la ejecución. La solución es crear 10 conexiones y ejecutar cada consulta en una conexión diferente. Esto se puede hacer automáticamente utilizando el conjunto de conexiones.

```
var pool = mysql.createPool({
  connectionLimit : 10,
  host             : 'example.org',
  user             : 'bobby',
  password         : 'pass',
  database         : 'schema'
});

for(var i=0;i<10;i++){
  pool.query('SELECT ` as example', function(err, rows, fields) {
    if (err) throw err;
    console.log(rows[0].example); //Show 1
```

```
});  
}
```

Se ejecutarán todas las 10 consultas en paralelo.

Cuando usas `pool` ya no necesitas la conexión. Puede consultar directamente la piscina. El módulo MySQL buscará la siguiente conexión gratuita para ejecutar su consulta.

segundo. Lograr multi-tenancy en el servidor de bases de datos con diferentes bases de datos alojadas en él.

La multipropiedad es un requisito común de las aplicaciones empresariales en la actualidad y no se recomienda crear un grupo de conexiones para cada base de datos en el servidor de bases de datos. Entonces, lo que podemos hacer es crear un grupo de conexión con el servidor de base de datos y luego cambiarlos entre las bases de datos alojadas en el servidor de base de datos a pedido.

Supongamos que nuestra aplicación tiene diferentes bases de datos para cada empresa alojada en el servidor de bases de datos. Nos conectaremos a la base de datos de la empresa respectiva cuando el usuario acceda a la aplicación. Aquí está el ejemplo de cómo hacer eso:

```
var pool = mysql.createPool({  
  connectionLimit : 10,  
  host             : 'example.org',  
  user             : 'bobby',  
  password         : 'pass'  
});  
  
pool.getConnection(function(err, connection){  
  if(err){  
    return cb(err);  
  }  
  connection.changeUser({database : "firm1"});  
  connection.query("SELECT * from history", function(err, data){  
    connection.release();  
    cb(err, data);  
  });  
});
```

Déjame desglosar el ejemplo:

Al definir la configuración del grupo, no le di el nombre de la base de datos, sino que solo le di un servidor de base de datos, es decir

```
{  
  connectionLimit : 10,  
  host             : 'example.org',  
  user             : 'bobby',  
  password         : 'pass'  
}
```

por eso, cuando queremos usar la base de datos específica en el servidor de la base de datos,

pedimos la conexión para golpear la base de datos mediante:

```
connection.changeUser({database : "firm1"});
```

Puede consultar la documentación oficial [aquí](#).

Conectarse a MySQL

Una de las maneras más fáciles de conectarse a MySQL es mediante el uso del módulo `mysql`. Este módulo maneja la conexión entre la aplicación Node.js y el servidor MySQL. Puedes instalarlo como cualquier otro módulo:

```
npm install --save mysql
```

Ahora tienes que crear una conexión `mysql`, que puedes consultar más tarde.

```
const mysql      = require('mysql');
const connection = mysql.createConnection({
  host      : 'localhost',
  user      : 'me',
  password  : 'secret',
  database  : 'database_schema'
});

connection.connect();

// Execute some query statements
// I.e. SELECT * FROM FOO

connection.end();
```

En el siguiente ejemplo, aprenderá a consultar el objeto de `connection`.

Consultar un objeto de conexión sin parámetros

Se envía la consulta como una cadena y en respuesta se recibe una llamada con la respuesta. La devolución de llamada le da `error`, matriz de `rows` y campos. Cada fila contiene toda la columna de la tabla devuelta. Aquí hay un fragmento de la siguiente explicación.

```
connection.query('SELECT name,email from users', function(err, rows, fields) {
  if (err) throw err;

  console.log('There are:', rows.length, ' users');
  console.log('First user name is:',rows[0].name)
});
```

Ejecutar una serie de consultas con una sola conexión de un grupo

Puede haber situaciones en las que haya configurado un grupo de conexiones MySQL, pero tiene una serie de consultas que le gustaría ejecutar en secuencia:

```
SELECT 1;
SELECT 2;
```

Se *podía* correr a continuación, utilizando `pool.query` [como se ve en otros lugares](#) , sin embargo, si sólo tiene una conexión libre en la piscina debe esperar hasta que la conexión esté disponible antes de poder ejecutar la segunda consulta.

Sin embargo, puede conservar una conexión activa del grupo y ejecutar tantas consultas como quiera usar con una sola conexión usando `pool.getConnection` :

```
pool.getConnection(function (err, conn) {
  if (err) return callback(err);

  conn.query('SELECT 1 AS seq', function (err, rows) {
    if (err) throw err;

    conn.query('SELECT 2 AS seq', function (err, rows) {
      if (err) throw err;

      conn.release();
      callback();
    });
  });
});
```

Nota: debe recordar `release` la conexión, de lo contrario, ¡hay una conexión MySQL menos disponible para el resto del grupo!

Para obtener más información sobre la agrupación de conexiones MySQL, [consulte la documentación de MySQL](#) .

Devuelve la consulta cuando se produce un error.

Puede adjuntar la consulta ejecutada a su objeto `err` cuando se produce un error:

```
var q = mysql.query('SELECT `name` FROM `pokedex` WHERE `id` = ?', [ 25 ], function (err, result) {
  if (err) {
    // Table 'test.pokedex' doesn't exist
    err.query = q.sql; // SELECT `name` FROM `pokedex` WHERE `id` = 25
    callback(err);
  }
  else {
    callback(null, result);
  }
});
```

Grupo de conexiones de exportación

```
// db.js

const mysql = require('mysql');
```

```
const pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bob',
  password        : 'secret',
  database        : 'my_db'
});

module.export = {
  getConnection: (callback) => {
    return pool.getConnection(callback);
  }
}
```

```
// app.js

const db = require('./db');

db.getConnection((err, conn) => {
  conn.query('SELECT something from sometable', (error, results, fields) => {
    // get the results
    conn.release();
  });
});
```

Lea Integración de MySQL en línea: <https://riptutorial.com/es/node-js/topic/1406/integracion-de-mysql>

Capítulo 64: Integración de pasaportes

Observaciones

La contraseña **siempre** debe estar oculta. Una forma sencilla de proteger contraseñas con **NodeJS** sería usar el módulo **bcrypt-nodejs** .

Examples

Empezando

El pasaporte se debe inicializar con el middleware `passport.initialize()` . Para utilizar las sesiones de inicio de sesión, se requiere el middleware `passport.session()` .

Tenga en cuenta que los métodos `passport.serialize()` y `passport.deserializeUser()` deben estar definidos. **Passport** serializará y deserializará las instancias de usuario hacia y desde la sesión

```
const express = require('express');
const session = require('express-session');
const passport = require('passport');
const cookieParser = require('cookie-parser');
const app = express();

// Required to read cookies
app.use(cookieParser());

passport.serializeUser(function(user, next) {
  // Serialize the user in the session
  next(null, user);
});

passport.deserializeUser(function(user, next) {
  // Use the previously serialized user
  next(null, user);
});

// Configuring express-session middleware
app.use(session({
  secret: 'The cake is a lie',
  resave: true,
  saveUninitialized: true
}));

// Initializing passport
app.use(passport.initialize());
app.use(passport.session());

// Starting express server on port 3000
app.listen(3000);
```

Autenticación local

El módulo **local de pasaporte** se utiliza para implementar una autenticación local.

Este módulo le permite autenticar con un nombre de usuario y contraseña en sus aplicaciones Node.js.

Registro del usuario:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

// A named strategy is used since two local strategy are used :
// one for the registration and the other to sign-in
passport.use('localSignup', new LocalStrategy({
  // Overriding defaults expected parameters,
  // which are 'username' and 'password'
  usernameField: 'email',
  passwordField: 'password',
  passReqToCallback: true // allows us to pass back the entire request to the callback
},
function(req, email, password, next) {
  // Check in database if user is already registered
  findUserByEmail(email, function(user) {
    // If email already exists, abort registration process and
    // pass 'false' to the callback
    if (user) return next(null, false);
    // Else, we create the user
    else {
      // Password must be hashed !
      let newUser = createUser(email, password);

      newUser.save(function() {
        // Pass the user to the callback
        return next(null, newUser);
      });
    }
  });
});
```

Iniciar sesión en el usuario:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

passport.use('localSignin', new LocalStrategy({
  usernameField : 'email',
  passwordField : 'password',
},
function(email, password, next) {
  // Find the user
  findUserByEmail(email, function(user) {
    // If user is not found, abort signing in process
    // Custom messages can be provided in the verify callback
    // to give the user more details concerning the failed authentication
    if (!user)
      return next(null, false, {message: 'This e-mail address is not associated with any account.'});
    // Else, we check if password is valid
    else {
```

```

        // If password is not correct, abort signing in process
        if (!isPasswordValid(password)) return next(null, false);
        // Else, pass the user to callback
        else return next(null, user);
    }
    });
});

```

Creando rutas:

```

// ...
app.use(passport.initialize());
app.use(passport.session());

// Sign-in route
// Passport strategies are middlewares
app.post('/login', passport.authenticate('localSignin', {
    successRedirect: '/me',
    failureRedirect: '/login'
}));

// Sign-up route
app.post('/register', passport.authenticate('localSignup', {
    successRedirect: '/',
    failureRedirect: '/signup'
}));

// Call req.logout() to log out
app.get('/logout', function(req, res) {
    req.logout();
    res.redirect('/');
});

app.listen(3000);

```

Autenticación de Facebook

El módulo **pasaporte-facebook** se utiliza para implementar una autenticación de **Facebook** . En este ejemplo, si el usuario no existe en el inicio de sesión, se crea.

Implementando la estrategia:

```

const passport = require('passport');
const FacebookStrategy = require('passport-facebook').Strategy;

// Strategy is named 'facebook' by default
passport.use({
    clientID: 'yourclientid',
    clientSecret: 'yourclientsecret',
    callbackURL: '/auth/facebook/callback'
},
// Facebook will send a token and user's profile
function(token, refreshToken, profile, next) {
    // Check in database if user is already registered
    findUserByFacebookId(profile.id, function(user) {
        // If user exists, returns his data to callback
        if (user) return next(null, user);
    });
});

```

```

    // Else, we create the user
    else {
        let newUser = createUserFromFacebook(profile, token);

        newUser.save(function() {
            // Pass the user to the callback
            return next(null, newUser);
        });
    }
});
});
});

```

Creando rutas:

```

// ...
app.use(passport.initialize());
app.use(passport.session());

// Authentication route
app.get('/auth/facebook', passport.authenticate('facebook', {
    // Ask Facebook for more permissions
    scope : 'email'
}));

// Called after Facebook has authenticated the user
app.get('/auth/facebook/callback',
    passport.authenticate('facebook', {
        successRedirect : '/me',
        failureRedirect : '/'
    }));

//...

app.listen(3000);

```

Autenticación de usuario-contraseña simple

En tus rutas / index.js

Aquí el `user` es el modelo para el usuario.

```

router.post('/login', function(req, res, next) {
    if (!req.body.username || !req.body.password) {
        return res.status(400).json({
            message: 'Please fill out all fields'
        });
    }

    passport.authenticate('local', function(err, user, info) {
        if (err) {
            console.log("ERROR : " + err);
            return next(err);
        }

        if(user) {

```

```

        console.log("User Exists!")
        //All the data of the user can be accessed by user.x
        res.json({"success" : true});
        return;
    } else {
        res.json({"success" : false});
        console.log("Error" + errorResponse());
        return;
    }
})(req, res, next);
});

```

Autenticación de Google Passport

Tenemos un módulo simple disponible en npm para el nombre de autenticación de las gafas **passport-google-oauth20**

Considere el siguiente ejemplo En este ejemplo, hemos creado una carpeta a saber, config que tiene el archivo passport.js y google.js en el directorio raíz. En tu app.js incluye lo siguiente

```

var express = require('express');
var session = require('express-session');
var passport = require('./config/passport'); // path where the passport file placed
var app = express();
passport(app);

```

// otro código para desactivar el servidor, manejador de errores

En el archivo passport.js en la carpeta de configuración, incluya el siguiente código

```

var passport = require ('passport'),
google = require('./google'),
User = require('./../model/user'); // User is the mongoose model

module.exports = function(app){
    app.use(passport.initialize());
    app.use(passport.session());
    passport.serializeUser(function(user, done){
        done(null, user);
    });
    passport.deserializeUser(function (user, done) {
        done(null, user);
    });
    google();
};

```

En el archivo google.js en la misma carpeta de configuración incluyen los siguientes

```

var passport = require('passport'),
GoogleStrategy = require('passport-google-oauth20').Strategy,
User = require('./../model/user');
module.exports = function () {
    passport.use(new GoogleStrategy({
        clientID: 'CLIENT ID',
        clientSecret: 'CLIENT SECRET',

```

```

        callbackURL: "http://localhost:3000/auth/google/callback"
    },
    function(accessToken, refreshToken, profile, cb) {
        User.findOne({ googleId : profile.id }, function (err, user) {
            if(err){
                return cb(err, false, {message : err});
            }else {
                if (user != '' && user != null) {
                    return cb(null, user, {message : "User "});
                } else {
                    var username = profile.displayName.split(' ');
                    var userData = new User({
                        name : profile.displayName,
                        username : username[0],
                        password : username[0],
                        facebookId : '',
                        googleId : profile.id,
                    });
                    // send email to user just in case required to send the newly created
                    // credentials to user for future login without using google login
                    userData.save(function (err, newuser) {
                        if (err) {
                            return cb(null, false, {message : err + " !!! Please try again"});
                        }else{
                            return cb(null, newuser);
                        }
                    });
                }
            }
        });
    }
});
});
});
};

```

Aquí, en este ejemplo, si el usuario no está en la base de datos, está creando un nuevo usuario en la base de datos para referencia local utilizando el nombre de campo googleId en el modelo de usuario.

Lea Integración de pasaportes en línea: <https://riptutorial.com/es/node-js/topic/7666/integracion-de-pasaportes>

Capítulo 65: Integración MSSQL

Introducción

Para integrar cualquier base de datos con nodejs, necesita un paquete de controladores o puede llamarlo módulo npm que le proporcionará una API básica para conectarse con la base de datos y realizar interacciones. Lo mismo ocurre con la base de datos mssql, aquí integraremos mssql con nodejs y realizaremos algunas consultas básicas en las tablas de SQL.

Observaciones

Hemos asumido que tendremos una instancia local del servidor de base de datos mssql ejecutándose en la máquina local. Puedes referir [este documento](#) para hacer lo mismo.

También asegúrese de que el usuario apropiado creado con privilegios agregados también.

Examples

Conectando con SQL vía. mssql npm module

Comenzaremos creando una aplicación de nodo simple con una estructura básica y luego conectando con la base de datos del servidor SQL local y realizando algunas consultas en esa base de datos.

Paso 1: Cree un directorio / carpeta con el nombre del proyecto que intenta crear. Inicialice una aplicación de nodo con el comando *npm init* que creará un package.json en el directorio actual.

```
mkdir mySqlApp
//folder created
cd mwSqlApp
//change to newly created directory
npm init
//answer all the question ..
npm install
//This will complete quickly since we have not added any packages to our app.
```

Paso 2: Ahora crearemos un archivo App.js en este directorio e instalaremos algunos paquetes que necesitaremos para conectarnos a sql db.

```
sudo gedit App.js
//This will create App.js file , you can use your fav. text editor :)
npm install --save mssql
//This will install the mssql package to you app
```

Paso 3: Ahora agregaremos una variable de configuración básica a nuestra aplicación que será utilizada por el módulo mssql para establecer una conexión.

```

console.log("Hello world, This is an app to connect to sql server.");
var config = {
  "user": "myusername", //default is sa
  "password": "yourStrong(!)Password",
  "server": "localhost", // for local machine
  "database": "staging", // name of database
  "options": {
    "encrypt": true
  }
}

sql.connect(config, err => {
  if(err){
    throw err ;
  }
  console.log("Connection Successful !");

  new sql.Request().query('select 1 as number', (err, result) => {
    //handle err
    console.dir(result)
    // This example uses callbacks strategy for getting results.
  })
});

sql.on('error', err => {
  // ... error handler
  console.log("Sql database connection error " ,err);
})

```

Paso 4: Este es el paso más fácil, donde iniciamos la aplicación y la aplicación se conectará al servidor de SQL e imprimirá algunos resultados simples.

```

node App.js
// Output :
// Hello world, This is an app to connect to sql server.
// Connection Successful !
// 1

```

Para usar promesas o asíncronos para la ejecución de consultas, consulte los documentos oficiales del paquete mssql:

- [Promesas](#)
- [Async / Await](#)

Lea Integración MSSQL en línea: <https://riptutorial.com/es/node-js/topic/9884/integracion-mssql>

Capítulo 66: Integración PostgreSQL

Examples

Conectarse a PostgreSQL

Usando el módulo npm de PostgreSQL .

instala dependencia desde npm

```
npm install pg --save
```

Ahora tienes que crear una conexión PostgreSQL, que puedes consultar más tarde.

Supongamos que Database_Name = estudiantes, Host = localhost y DB_User = postgres

```
var pg = require("pg")
var connectionString = "pg://postgres:postgres@localhost:5432/students";
var client = new pg.Client(connectionString);
client.connect();
```

Consulta con objeto de conexión

Si desea utilizar el objeto de conexión para la base de datos de consultas, puede utilizar este código de ejemplo.

```
var queryString = "SELECT name, age FROM students " ;
var query = client.query(queryString);

query.on("row", (row, result)=> {
  result.addRow(row);
});

query.on("end", function (result) {
  //LOGIC
});
```

Lea Integración PostgreSQL en línea: <https://riptutorial.com/es/node-js/topic/7706/integracion-postgresql>

Capítulo 67: Interactuando con la consola

Sintaxis

- `console.log ([data] [, ...])`
- `console.error ([data] [, ...])`
- `console.time (etiqueta)`
- `console.timeEnd (etiqueta)`

Examples

Explotación florestal

Módulo de consola

Similar al entorno de navegación de JavaScript, node.js proporciona un módulo de **consola** que brinda posibilidades simples de registro y depuración.

Los métodos más importantes proporcionados por el módulo de `console.log` son `console.log`, `console.error` y `console.time`. Pero hay varios otros como `console.info`.

console.log

Los parámetros se imprimirán en la salida estándar (`stdout`) con una nueva línea.

```
console.log('Hello World');
```

```
> console.log('Hello World')
Hello World
```

console.error

Los parámetros se imprimirán al error estándar (`stderr`) con una nueva línea.

```
console.error('Oh, sorry, there is an error.');
```

```
> console.error("Oh, sorry, error");
Oh, sorry, error
```

console.time, console.timeEnd

`console.time` inicia un temporizador con una etiqueta única que se puede usar para calcular la duración de una operación. Cuando llama a `console.timeEnd` con la misma etiqueta, el temporizador se detiene e imprime el tiempo transcurrido en milisegundos hasta la `stdout`.

```
> console.time("label");
undefined
> console.timeEnd("label");
label: 9297.320ms
```

Módulo de proceso

Es posible utilizar el módulo de **proceso** para escribir **directamente** en la salida estándar de la consola. Por lo tanto existe el método `process.stdout.write`. A diferencia de `console.log` este método no agrega una nueva línea antes de su salida.

Entonces, en el siguiente ejemplo, el método se llama dos veces, pero no se agrega una nueva línea entre sus salidas.

```
> process.stdout.write("123");process.stdout.write("456");
123456true
```

Formateo

Uno puede usar **códigos de terminal (control)** para emitir comandos específicos como cambiar colores o posicionar el cursor.

```
> console.log("\033[31mThis will be red");
This will be red
```

General

Efecto	Código
Reiniciar	\033[0m
Hicolor	\033[1m
Subrayar	\033[4m
Inverso	\033[7m

Colores de fuente

Efecto	Código
Negro	\033[30m
rojo	\033[31m
Verde	\033[32m

Efecto	Código
Amarillo	\033[33m
Azul	\033[34m
Magenta	\033[35m
Cian	\033[36m
Blanco	\033[37m

Colores de fondo

Efecto	Código
Negro	\033[40m
rojo	\033[41m
Verde	\033[42m
Amarillo	\033[43m
Azul	\033[44m
Magenta	\033[45m
Cian	\033[46m
Blanco	\033[47m

Lea Interactuando con la consola en línea: <https://riptutorial.com/es/node-js/topic/5935/interactuando-con-la-consola>

Capítulo 68: Inyección de dependencia

Examples

¿Por qué usar la inyección de dependencia

1. **Rápido proceso de desarrollo**
2. **Desacoplamiento**
3. **Prueba de unidad de escritura**

Rápido proceso de desarrollo

Cuando se usa el desarrollador de nodos de inyección de dependencias, se puede acelerar su proceso de desarrollo porque después de DI hay menos conflicto de código y es fácil administrar todos los módulos.

Desacoplamiento

Los módulos se vuelven menos acoplados, entonces es fácil de mantener.

Prueba de unidad de escritura

Las dependencias codificadas pueden pasarlas al módulo y, a continuación, es fácil escribir la prueba de unidad para cada módulo.

Lea Inyección de dependencia en línea: <https://riptutorial.com/es/node-js/topic/7681/inyeccion-de-dependencia>

Capítulo 69: Koa Framework v2

Examples

Hola mundo ejemplo

```
const Koa = require('koa')

const app = new Koa()

app.use(async ctx => {
  ctx.body = 'Hello World'
})

app.listen(8080)
```

Manejo de errores utilizando middleware.

```
app.use(async (ctx, next) => {
  try {
    await next() // attempt to invoke the next middleware downstream
  } catch (err) {
    handleError(err, ctx) // define your own error handling function
  }
})
```

Lea Koa Framework v2 en línea: <https://riptutorial.com/es/node-js/topic/6730/koa-framework-v2>

Capítulo 70: La comunicación arduino con nodeJs.

Introducción

Manera de mostrar cómo Node.Js puede comunicarse con Arduino Uno.

Examples

Comunicación del nodo Js con Arduino a través de serialport.

Codigo js del nodo

La muestra para iniciar este tema es el servidor Node.js que se comunica con Arduino a través de serialport.

```
npm install express --save
npm install serialport --save
```

Ejemplo de aplicación.js:

```
const express = require('express');
const app = express();
var SerialPort = require("serialport");

var port = 3000;

var arduinoCOMPort = "COM3";

var arduinoSerialPort = new SerialPort(arduinoCOMPort, {
  baudrate: 9600
});

arduinoSerialPort.on('open',function() {
  console.log('Serial Port ' + arduinoCOMPort + ' is opened.');
```

```
});
```

```
app.get('/', function (req, res) {
```

```
    return res.send('Working');
```

```
})
```

```
app.get('/:action', function (req, res) {
```

```
    var action = req.params.action || req.param('action');
```

```
    if(action == 'led'){
```

```
        arduinoSerialPort.write("w");
```

```
        return res.send('Led light is on!');
```

```

    }
    if(action == 'off') {
        arduinoSerialPort.write("t");
        return res.send("Led light is off!");
    }

    return res.send('Action: ' + action);

});

app.listen(port, function () {
    console.log('Example app listening on port http://0.0.0.0:' + port + '!');
});

```

Iniciando el servidor express de muestra:

```
node app.js
```

Código arduino

```

// the setup function runs once when you press reset or power the board
void setup() {
    // initialize digital pin LED_BUILTIN as an output.

    Serial.begin(9600); // Begen listening on port 9600 for serial

    pinMode(LED_BUILTIN, OUTPUT);

    digitalWrite(LED_BUILTIN, LOW);
}

// the loop function runs over and over again forever
void loop() {

    if(Serial.available() > 0) // Read from serial port
    {
        char ReaderFromNode; // Store current character
        ReaderFromNode = (char) Serial.read();
        convertToState(ReaderFromNode); // Convert character to state
    }
    delay(1000);
}

void convertToState(char chr) {
    if(chr=='o'){
        digitalWrite(LED_BUILTIN, HIGH);
        delay(100);
    }
    if(chr=='f'){
        digitalWrite(LED_BUILTIN, LOW);
        delay(100);
    }
}

```

Empezando

1. Conecta el arduino a tu maquina.
2. Iniciar el servidor

Controlar la construcción en led vía nodo js servidor expreso.

Para encender el led:

```
http://0.0.0.0:3000/led
```

Para apagar el led:

```
http://0.0.0.0:3000/off
```

Lea La comunicación arduino con nodeJs. en línea: <https://riptutorial.com/es/node-js/topic/10509/la-comunicacion-arduino-con-nodejs->

Capítulo 71: Localización Nod JS

Introducción

Es muy fácil de mantener la localización de los nodejs express.

Examples

utilizando el módulo i18n para mantener la localización en la aplicación node js

Módulo de traducción simple y ligero con almacenamiento dinámico de json. Es compatible con las aplicaciones plain vanilla node.js y debe funcionar con cualquier marco (como Express, Restify y probablemente más) que expone un método app.use () que pasa objetos res y req. Utiliza la sintaxis __ ('...') común en aplicaciones y plantillas. Almacena archivos de idioma en archivos json compatibles con el formato webtranslateit json. Agrega nuevas cadenas sobre la marcha cuando se usan por primera vez en su aplicación. No se necesita un análisis adicional.

Expressa + i18n-node + cookieParser y evita problemas de concurrencia

```
// usual requirements
var express = require('express'),
    i18n = require('i18n'),
    app = module.exports = express();

i18n.configure({
  // setup some locales - other locales default to en silently
  locales: ['en', 'ru', 'de'],

  // sets a custom cookie name to parse locale settings from
  cookie: 'yourcookienam',

  // where to store json files - defaults to './locales'
  directory: __dirname + '/locales'
});

app.configure(function () {
  // you will need to use cookieParser to expose cookies to req.cookies
  app.use(express.cookieParser());

  // i18n init parses req for language headers, cookies, etc.
  app.use(i18n.init);
});

// serving homepage
app.get('/', function (req, res) {
  res.send(res.__('Hello World'));
});

// starting server
if (!module.parent) {
```

```
app.listen(3000);  
}
```

Lea Localización Nodo JS en línea: <https://riptutorial.com/es/node-js/topic/9594/localizacion-nodo-js>

Capítulo 72: Lodash

Introducción

Lodash es una útil biblioteca de utilidades de JavaScript.

Examples

Filtrar una colección

El fragmento de código a continuación muestra las diversas formas en que puede filtrar en una matriz de objetos usando lodash.

```
let lodash = require('lodash');

var countries = [
  {"key": "DE", "name": "Deutschland", "active": false},
  {"key": "ZA", "name": "South Africa", "active": true}
];

var filteredByFunction = lodash.filter(countries, function (country) {
  return country.key === "DE";
});
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByObjectProperties = lodash.filter(countries, { "key": "DE" });
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByProperties = lodash.filter(countries, ["key", "ZA"]);
// => [{"key": "ZA", "name": "South Africa"}];

var filteredByProperty = lodash.filter(countries, "active");
// => [{"key": "ZA", "name": "South Africa"}];
```

Lea Lodash en línea: <https://riptutorial.com/es/node-js/topic/9161/lodash>

Capítulo 73: Loopback - Conector basado en REST

Introducción

Conectores basados en reposo y cómo tratar con ellos. Todos sabemos que Loopback no proporciona elegancia a las conexiones basadas en REST.

Examples

Agregar un conector basado en web

```
// Este ejemplo obtiene la respuesta de iTunes
{
  "descanso": {
    "nombre": "resto",
    "conector": "resto",
    "depurar": verdadero,
    "opciones": {
      "useQuerystring": verdadero,
      "tiempo de espera": 10000,
      "encabezados": {
        "acepta": "aplicación / json",
        "tipo de contenido": "aplicación / json"
      }
    }
  }
}
"operaciones": [
  {
    "modelo": {
      "método": "OBTENER",
      "url": "https://itunes.apple.com/search",
      "consulta": {
        "term": "{keyword}",
        "country": "{country = IN}",
        "media": "{itemType = music}",
        "límite": "{límite = 10}",
        "explícito": "falso"
      }
    }
  }
]
"funciones": {
  "buscar": [
    "palabra clave",
    "país",
    "tipo de artículo",
    "límite"
  ]
}
}
{
  "modelo": {
    "método": "OBTENER",
    "url": "https://itunes.apple.com/lookup",
    "consulta": {
```

```
        "yo si}"
    }
}
"funciones": {
  "findById": [
    "carné de identidad"
  ]
}
]
}
}
```

Lea Loopback - Conector basado en REST en línea: <https://riptutorial.com/es/node-js/topic/9234/loopback---conector-basado-en-rest>

Capítulo 74: Manejo de excepciones

Examples

Manejo de excepciones en Node.js

Node.js tiene 3 formas básicas de manejar excepciones / errores:

1. **tratar - atrapar** bloque
2. **error** como el primer argumento de una `callback`
3. `emit` un evento de **error** utilizando `eventEmitter`

try-catch se utiliza para capturar las excepciones generadas desde la ejecución del código síncrono. Si la persona que llama (o la persona que llama, ...) usó try / catch, entonces pueden detectar el error. Si ninguna de las personas que llamaron tuvo un intento de captura, el programa se bloquea.

Si se utiliza try-catch en una operación asíncrona y se generó una excepción a partir de la devolución de llamada del método `async`, entonces no se detectará mediante try-catch. Para capturar una excepción de la devolución de llamada de operación asíncrona, se prefiere usar *promesas* .

Ejemplo para entenderlo mejor.

```
// ** Example - 1 **
function doSomeSynchronousOperation(req, res) {
  if(req.body.username === ''){
    throw new Error('User Name cannot be empty');
  }
  return true;
}

// calling the method above
try {
  // synchronous code
  doSomeSynchronousOperation(req, res)
} catch(e) {
  //exception handled here
  console.log(e.message);
}

// ** Example - 2 **
function doSomeAsynchronousOperation(req, res, cb) {
  // imitating async operation
  return setTimeout(function(){
    cb(null, []);
  },1000);
}

try {
  // asynchronous code
  doSomeAsynchronousOperation(req, res, function(err, rs){
    throw new Error("async operation exception");
  })
}
```

```
} catch(e) {  
    // Exception will not get handled here  
    console.log(e.message);  
}  
// The exception is unhandled and hence will cause application to break
```

Las devoluciones de llamada se utilizan principalmente en Node.js ya que la devolución de llamada entrega un evento de forma asíncrona. El usuario le pasa una función (la devolución de llamada), y la invoca más tarde cuando finaliza la operación asíncrona. El patrón habitual es que la devolución de llamada se invoca como *devolución de llamada (error, resultado)*, donde solo uno de error y resultado no es nulo, dependiendo de si la operación se realizó correctamente o no.

```
function doSomeAsynchronousOperation(req, res, callback) {  
    setTimeout(function(){  
        return callback(new Error('User Name cannot be empty'));  
    }, 1000);  
    return true;  
}  
  
doSomeAsynchronousOperation(req, res, function(err, result) {  
    if (err) {  
        //exception handled here  
        console.log(err.message);  
    }  
  
    //do some stuff with valid data  
});
```

emiten Para los casos más complicados, en lugar de utilizar una devolución de llamada, la propia función puede devolver un objeto EventEmitter, y se esperaría que la persona que llama para escuchar los eventos de error en el emisor.

```
const EventEmitter = require('events');  
  
function doSomeAsynchronousOperation(req, res) {  
    let myEvent = new EventEmitter();  
  
    // runs asynchronously  
    setTimeout(function(){  
        myEvent.emit('error', new Error('User Name cannot be empty'));  
    }, 1000);  
  
    return myEvent;  
}  
  
// Invoke the function  
let event = doSomeAsynchronousOperation(req, res);  
  
event.on('error', function(err) {  
    console.log(err);  
});  
  
event.on('done', function(result) {  
    console.log(result); // true  
});
```

Gestión de excepciones no gestionadas

Debido a que Node.js se ejecuta en un solo proceso, las excepciones no detectadas son un problema que se debe tener en cuenta al desarrollar aplicaciones.

Manejo silencioso de excepciones

La mayoría de las personas permiten que los servidores node.js traguen silenciosamente los errores.

- Manejo silencioso de la excepción.

```
process.on('uncaughtException', function (err) {  
  console.log(err);  
});
```

Esto es malo , funcionará pero:

- La causa raíz seguirá siendo desconocida, por lo que no contribuirá a la resolución de lo que causó la excepción (error).
- En caso de que la conexión de la base de datos (grupo) se cierre por algún motivo, esto dará lugar a una constante propagación de errores, lo que significa que el servidor se ejecutará pero no se volverá a conectar a db.

Volviendo al estado inicial

En caso de una "excepción no captada", es bueno reiniciar el servidor y devolverlo a su **estado inicial** , donde sabemos que funcionará. Se registra una excepción, la aplicación finaliza pero como se ejecutará en un contenedor que se asegurará de que el servidor se está ejecutando, lograremos el reinicio del servidor (volviendo al estado de funcionamiento inicial).

- Instalación de forever (u otra herramienta CLI para asegurarse de que el servidor de nodos se ejecuta continuamente)

```
npm install forever -g
```

- Iniciando el servidor en siempre

```
forever start app.js
```

La razón por la cual se inició y la razón por la que usamos para siempre es después de que el servidor se **termina**, el proceso iniciará nuevamente el servidor.

- Reiniciando el servidor

```
process.on('uncaughtException', function (err) {
  console.log(err);

  // some logging mechanisam
  // ....

  process.exit(1); // terminates process
});
```

En una nota al margen, también había una forma de manejar las excepciones con **Clústeres y Dominios** .

Los dominios están en desuso más información [aquí](#) .

Errores y promesas

Las promesas manejan los errores de manera diferente al código sincrónico o de devolución de llamada.

```
const p = new Promise(function (resolve, reject) {
  reject(new Error('Oops'));
});

// anything that is `reject`ed inside a promise will be available through catch
// while a promise is rejected, `.then` will not be called
p
  .then(() => {
    console.log("won't be called");
  })
  .catch(e => {
    console.log(e.message); // output: Oops
  })
  // once the error is caught, execution flow resumes
  .then(() => {
    console.log('hello!'); // output: hello!
  });
```

Actualmente, los errores que se lanzan en una promesa que no se captura provocan que el error se trague, lo que puede dificultar la localización del error. Esto se puede [resolver](#) utilizando herramientas de [alineación](#) como [eslint](#) o asegurándose de que siempre tenga una cláusula `catch` .

Este comportamiento está [en desuso en el nodo 8](#) a favor de terminar el proceso del nodo.

Lea Manejo de excepciones en línea: <https://riptutorial.com/es/node-js/topic/2819/manejo-de-excepciones>

Capítulo 75: Manejo de solicitud POST en Node.js

Observaciones

Node.js utiliza [secuencias](#) para manejar los datos entrantes.

Citando de los documentos,

Un flujo es una interfaz abstracta para trabajar con datos de transmisión en Node.js. El módulo de flujo proporciona una API base que facilita la creación de objetos que implementan la interfaz de flujo.

Para manejar el cuerpo de la solicitud de una solicitud POST, use el objeto de `request`, que es un flujo legible. Los flujos de datos se emiten como eventos de `data` en el objeto de `request`.

```
request.on('data', chunk => {
  buffer += chunk;
});
request.on('end', () => {
  // POST request body is now available as `buffer`
});
```

Simplemente cree una cadena de búfer vacía y agregue los datos del búfer como se recibieron a través de `data` eventos de `data`.

NOTA

1. Los datos de búfer recibidos en eventos de `data` son de tipo [Búfer](#)
2. Cree una nueva cadena de búfer para recopilar datos almacenados en búfer de los eventos de datos **para cada solicitud**, es decir, cree una cadena de `buffer` dentro del controlador de solicitudes.

Examples

Ejemplo de servidor node.js que solo maneja solicitudes POST

```
'use strict';

const http = require('http');

const PORT = 8080;
const server = http.createServer((request, response) => {
  let buffer = '';
  request.on('data', chunk => {
    buffer += chunk;
  });
  request.on('end', () => {
```

```
const responseString = `Received string ${buffer}`;  
console.log(`Responding with: ${responseString}`);  
response.writeHead(200, "Content-Type: text/plain");  
response.end(responseString);  
});  
}).listen(PORT, () => {  
  console.log(`Listening on ${PORT}`);  
});
```

Lea Manejo de solicitud POST en Node.js en línea: <https://riptutorial.com/es/node-js/topic/5676/manejo-de-solicitud-post-en-node-js>

Capítulo 76: Mantener una aplicación de nodo constantemente en ejecución

Examples

Usa PM2 como administrador de procesos

PM2 te permite ejecutar tus scripts de nodejs para siempre. En caso de que su aplicación falle, PM2 también la reiniciará por usted.

Instale PM2 globalmente para administrar sus instancias de nodejs

```
npm install pm2 -g
```

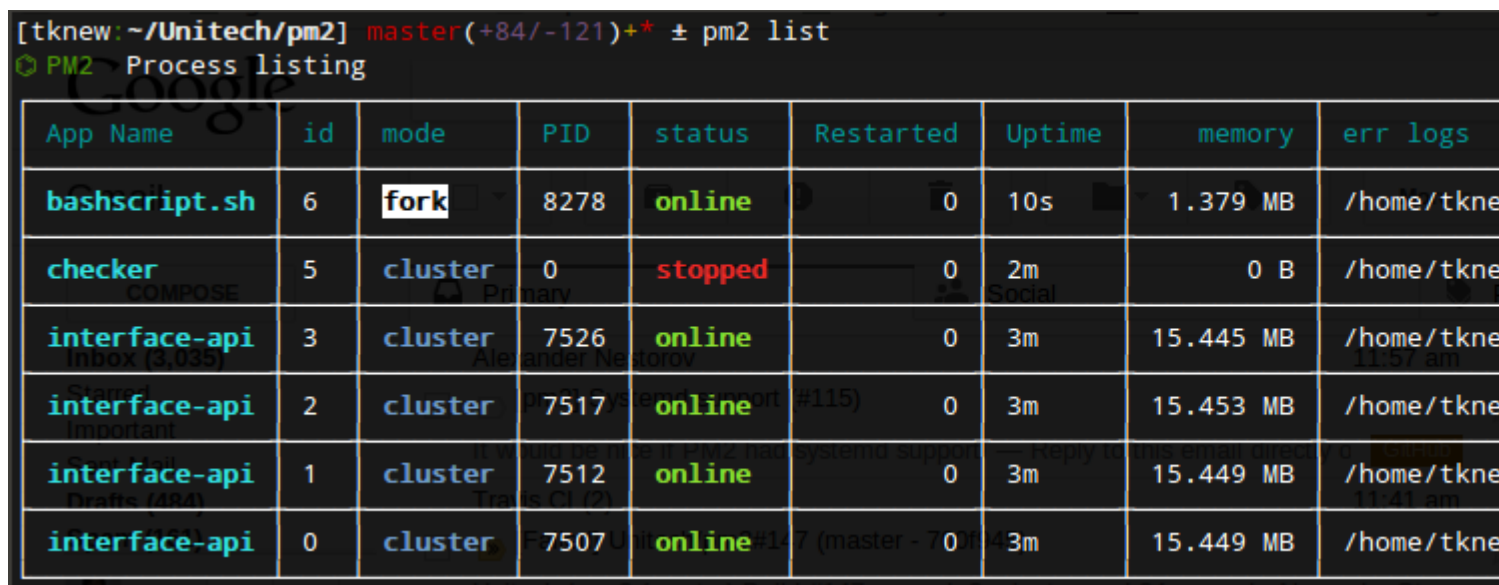
Navegue hasta el directorio en el que reside su script de nodejs y ejecute el siguiente comando cada vez que desee iniciar una instancia de nodejs para que sea supervisada por pm2:

```
pm2 start server.js --name "app1"
```

Comandos útiles para monitorear el proceso.

1. Listar todas las instancias de nodejs gestionadas por pm2

```
pm2 list
```



App Name	id	mode	PID	status	Restarted	Uptime	memory	err logs
bashscript.sh	6	fork	8278	online	0	10s	1.379 MB	/home/tkne
checker	5	cluster	0	stopped	0	2m	0 B	/home/tkne
interface-api	3	cluster	7526	online	0	3m	15.445 MB	/home/tkne
interface-api	2	cluster	7517	online	0	3m	15.453 MB	/home/tkne
interface-api	1	cluster	7512	online	0	3m	15.449 MB	/home/tkne
interface-api	0	cluster	7507	online	0	43m	15.449 MB	/home/tkne

2. Detener una instancia de nodejs particular

```
pm2 stop <instance named>
```

3. Eliminar una instancia de nodejs particular

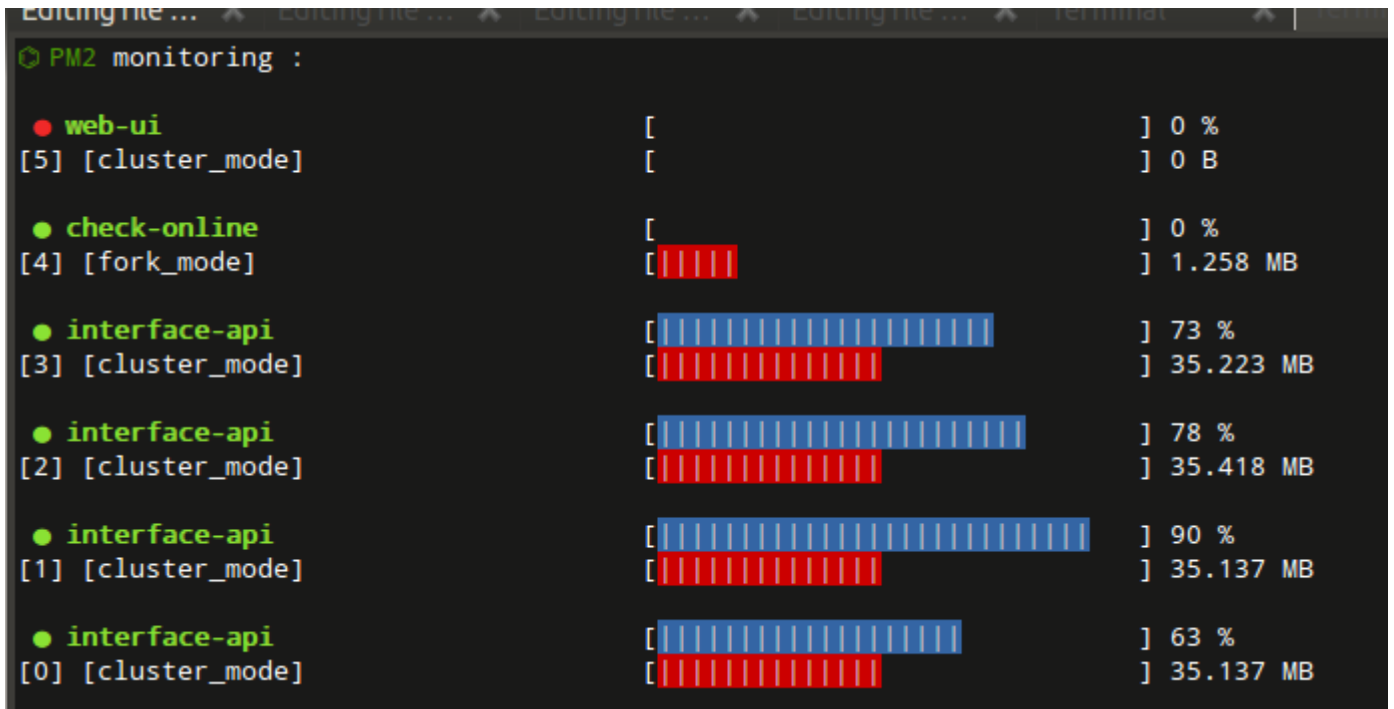
```
pm2 delete <instance name>
```

4. Reinicie una instancia de nodejs particular

```
pm2 restart <instance name>
```

5. Monitorizando todas las instancias de nodejs

```
pm2 monit
```



6. Parada pm2

```
pm2 kill
```

7. A diferencia de reiniciar, que mata y reinicia el proceso, la recarga logra una recarga de tiempo de inactividad de 0 segundos

```
pm2 reload <instance name>
```

8. Ver los registros

```
pm2 logs <instance_name>
```

Ejecutando y deteniendo un demonio de Forever

Para iniciar el proceso:

```
$ forever start index.js
warn:    --minUptime not set. Defaulting to: 1000ms
warn:    --spinSleepTime not set. Your script will exit if it does not stay up for at least
1000ms
info:    Forever processing file: index.js
```

Lista de instancias en ejecución para siempre:

```
$ forever list
info:    Forever processes running

|data: | index | uid | command          | script          |forever pid|id   | logfile
|uptime      |
|-----|-----|-----|-----|-----|-----|-----|-----|
---|-----|
|data: | [0]    |f4Kt | /usr/bin/nodejs  | src/index.js|2131      |
2146|/root/.forever/f4Kt.log | 0:0:0:11.485 |
```

Detener el primer proceso:

```
$ forever stop 0

$ forever stop 2146

$ forever stop --uid f4Kt

$ forever stop --pidFile 2131
```

Carrera continua con nohup

Una alternativa para siempre en Linux es nohup.

Para iniciar una instancia nohup

1. CD a la ubicación de la carpeta `app.js` o `www`
2. ejecutar `nohup nodejs app.js &`

Matar el proceso

1. ejecuta `ps -ef|grep nodejs`
2. `kill -9 <the process number>`

Proceso de gestión con Forever

Instalación

```
npm install forever -g
cd /node/project/directory
```

Usos

```
forever start app.js
```

Lea [Mantener una aplicación de nodo constantemente en ejecución en línea](https://riptutorial.com/es/node-js/topic/2820/mantener-una-aplicacion-de-nodo-constantemente-en-ejecucion):

<https://riptutorial.com/es/node-js/topic/2820/mantener-una-aplicacion-de-nodo-constantemente-en-ejecucion>

Capítulo 77: Marcos de plantillas

Examples

Nunjucks

Motor del lado del servidor con herencia de bloque, autoescape, macros, control asíncrono y más. Muy inspirado en jinja2, muy similar a Twig (php).

Docs - <http://mozilla.github.io/nunjucks/>

Instalar - `npm i nunjucks`

Uso básico con [Express](#) a continuación.

app.js

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

var app = express();
app.use(express.static('/public'));

// Apply nunjucks and add custom filter and function (for example).
var env = nunjucks.configure(['views/'], { // set folders with templates
  autoescape: true,
  express: app
});
env.addFilter('myFilter', function(obj, arg1, arg2) {
  console.log('myFilter', obj, arg1, arg2);
  // Do smth with obj
  return obj;
});
env.addGlobal('myFunc', function(obj, arg1) {
  console.log('myFunc', obj, arg1);
  // Do smth with obj
  return obj;
});

app.get('/', function(req, res){
  res.render('index.html', {title: 'Main page'});
});

app.get('/foo', function(req, res){
  res.locals.smthVar = 'This is Sparta!';
  res.render('foo.html', {title: 'Foo page'});
});

app.listen(3000, function() {
  console.log('Example app listening on port 3000...');
});
```

/views/index.html

```
<html>
<head>
  <title>Nunjucks example</title>
</head>
<body>
  {% block content %}
  {{title}}
  {% endblock %}
</body>
</html>
```

/views/foo.html

```
{% extends "index.html" %}

{# This is comment #}
{% block content %}
  <h1>{{title}}</h1>
  {# apply custom function and next build-in and custom filters #}
  {{ myFunc(smithVar) | lower | myFilter(5, 'abc') }}
{% endblock %}
```

Lea Marcos de plantillas en línea: <https://riptutorial.com/es/node-js/topic/5885/marcos-de-plantillas>

Capítulo 78: Marcos de pruebas unitarias

Examples

Mocha síncrona

```
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function() {
      expect([ 1, 2, 3 ].length).to.be.equal(3)
    })
  })
})
```

Mocha asíncrono (callback)

```
var expect = require("chai").expect;
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function(done) {
      testSomething(err => {
        expect(err).to.not.be.equal(null)
        done()
      })
    })
  })
})
```

Mocha asíncrona (Promesa)

```
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function() {
      return doSomething().then(result => {
        expect(result).to.be.equal('hello world')
      })
    })
  })
})
```

Mocha Asíncrono (asíncrono / await)

```
const { expect } = require('chai')

describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', async function() {
      const result = await answerToTheUltimateQuestion()
      expect(result).to.be.equal(42)
    })
  })
})
```

```
})
```

Lea Marcos de pruebas unitarias en línea: <https://riptutorial.com/es/node-js/topic/6731/marcos-de-pruebas-unitarias>

Capítulo 79: Módulo de cluster

Sintaxis

- `const cluster = require ("cluster")`
- `cluster.fork ()`
- `cluster.isMaster`
- `cluster.isWorker`
- `cluster.schedulingPolicy`
- `cluster.setupMaster (configuración)`
- `cluster.settings`
- `cluster.worker // in worker`
- `cluster.workers // en master`

Observaciones

Tenga en cuenta que `cluster.fork()` genera un proceso hijo que comienza a ejecutar la secuencia de comandos actual desde el principio, en contraste con la llamada al sistema de `fork()` en C, que clona el proceso actual y continúa a partir de la instrucción después de la llamada del sistema tanto en el padre como en el servidor. proceso hijo

La documentación de Node.js tiene una guía más completa para los clústeres [aquí](#)

Examples

Hola Mundo

Este es tu `cluster.js` :

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  require('./server.js')();
}
```

Este es tu `server.js` principal:

```
const http = require('http');

function startServer() {
  const server = http.createServer((req, res) => {
    res.writeHead(200);
    res.end('Hello Http');
  });

  server.listen(3000);
}

if(!module.parent) {
  // Start server if file is run directly
  startServer();
} else {
  // Export server, if file is referenced via cluster
  module.exports = startServer;
}
```

En este ejemplo, alojamos un servidor web básico; sin embargo, activamos trabajadores (procesos secundarios) utilizando el módulo de **clúster** incorporado. La cantidad de forker de procesos depende de la cantidad de núcleos de CPU disponibles. Esto permite que una aplicación Node.js aproveche las CPU de varios núcleos, ya que una sola instancia de Node.js se ejecuta en un solo hilo. La aplicación ahora compartirá el puerto 8000 en todos los procesos. Las cargas se distribuirán automáticamente entre los trabajadores utilizando el método Round-Robin de forma predeterminada.

Ejemplo de cluster

Una sola instancia de `Node.js` ejecuta en un solo hilo. Para aprovechar los sistemas de múltiples núcleos, la aplicación se puede iniciar en un clúster de procesos Node.js para manejar la carga.

El módulo de `cluster` permite crear fácilmente procesos secundarios que comparten todos los puertos del servidor.

El siguiente ejemplo crea el proceso hijo del trabajador en el proceso principal que maneja la carga a través de múltiples núcleos.

Ejemplo

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length; //number of CPUs

if (cluster.isMaster) {
  // Fork workers.
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork(); //creating child process
  }

  //on exit of cluster
  cluster.on('exit', (worker, code, signal) => {
```

```
    if (signal) {
      console.log(`worker was killed by signal: ${signal}`);
    } else if (code !== 0) {
      console.log(`worker exited with error code: ${code}`);
    } else {
      console.log('worker success!');
    }
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(3000);
}
```

Lea Módulo de cluster en línea: <https://riptutorial.com/es/node-js/topic/2817/modulo-de-cluster>

Capítulo 80: Multihilo

Introducción

Node.js ha sido diseñado para ser de un solo hilo. Así que para todos los propósitos prácticos, las aplicaciones que se inician con Node se ejecutarán en un solo hilo.

Sin embargo, Node.js se ejecuta en varios subprocesos. Las operaciones de E / S y similares se ejecutarán desde un grupo de subprocesos. Además, cualquier instancia de una aplicación de nodo se ejecutará en un subproceso diferente, por lo tanto, para ejecutar aplicaciones de subprocesos múltiples, se inician varias instancias.

Observaciones

Comprender el [bucle de eventos](#) es importante para comprender cómo y por qué utilizar varios subprocesos.

Examples

Racimo

El módulo de `cluster` permite iniciar la misma aplicación varias veces.

El agrupamiento es deseable cuando las diferentes instancias tienen el mismo flujo de ejecución y no dependen unas de otras. En este escenario, tiene un maestro que puede iniciar las horquillas y las horquillas (o hijos). Los niños trabajan de forma independiente y tienen su único espacio de Ram y Event Loop.

La configuración de clústeres puede ser beneficiosa para sitios web / API. Cualquier hilo puede servir a cualquier cliente, ya que no depende de otros hilos. Una base de datos (como Redis) se usaría para compartir cookies, ya que las **variables no se pueden compartir** entre los hilos.

```
// runs in each instance
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;

console.log('I am always called');

if (cluster.isMaster) {
  // runs only once (within the master);
  console.log('I am the master, launching workers!');
  for(var i = 0; i < numCPUs; i++) cluster.fork();
} else {
  // runs in each fork
  console.log('I am a fork!');

  // here one could start, as an example, a web server
```

```
}  
  
console.log('I am always called as well');
```

Proceso infantil

Los procesos secundarios son el camino a seguir cuando uno quiere ejecutar procesos de forma independiente con diferentes inicializaciones e inquietudes. Al igual que las horquillas en los grupos, un `child_process` ejecuta en su hilo, pero a diferencia de las horquillas, tiene una manera de comunicarse con su padre.

La comunicación va en ambos sentidos, por lo que los padres y el niño pueden escuchar los mensajes y enviar mensajes.

Padre (../parent.js)

```
var child_process = require('child_process');  
console.log('[Parent]', 'inititalize');
```



```
var child1 = child_process.fork(__dirname + '/child');  
child1.on('message', function(msg) {  
    console.log('[Parent]', 'Answer from child: ', msg);  
});
```



```
// one can send as many messages as one want  
child1.send('Hello'); // Hello to you too :)  
child1.send('Hello'); // Hello to you too :)
```



```
// one can also have multiple children  
var child2 = child_process.fork(__dirname + '/child');
```

Niño (../child.js)

```
// here would one initialize this child  
// this will be executed only once  
console.log('[Child]', 'inititalize');
```



```
// here one listens for new tasks from the parent  
process.on('message', function(messageFromParent) {  
  
    //do some intense work here  
    console.log('[Child]', 'Child doing some intense work');
```



```
    if(messageFromParent == 'Hello') process.send('Hello to you too :)');  
    else process.send('what?');
```



```
})
```

Junto al mensaje, se pueden escuchar **muchos eventos** como 'error', 'conectado' o 'desconectar'.

Iniciar un proceso hijo tiene un cierto costo asociado. Uno querría engendrar la menor cantidad posible de ellos.

Lea Multihilo en línea: <https://riptutorial.com/es/node-js/topic/10592/multihilo>

Capítulo 81: N-API

Introducción

El N-API es una nueva y mejor manera de crear un módulo nativo para NodeJS. N-API se encuentra en una etapa temprana, por lo que puede tener documentación inconsistente.

Examples

Hola a N-API

Este módulo registra la función de saludo en el módulo de saludo. La función hello imprime Hello world en la consola con `printf` y devuelve 1373 de la función nativa al llamador javascript.

```
#include <node_api.h>
#include <stdio.h>

napi_value say_hello(napi_env env, napi_callback_info info)
{
    napi_value retval;

    printf("Hello world\n");

    napi_create_number(env, 1373, &retval);

    return retval;
}

void init(napi_env env, napi_value exports, napi_value module, void* priv)
{
    napi_status status;
    napi_property_descriptor desc = {
        /*
         * String describing the key for the property, encoded as UTF8.
         */
        .utf8name = "hello",
        /*
         * Set this to make the property descriptor object's value property
         * to be a JavaScript function represented by method.
         * If this is passed in, set value, getter and setter to NULL (since these members
         won't be used).
         */
        .method = say_hello,
        /*
         * A function to call when a get access of the property is performed.
         * If this is passed in, set value and method to NULL (since these members won't be
         used).
         * The given function is called implicitly by the runtime when the property is
         accessed
         * from JavaScript code (or if a get on the property is performed using a N-API call).
         */
        .getter = NULL,
        /*
         */
    }
```

```

    * A function to call when a set access of the property is performed.
    * If this is passed in, set value and method to NULL (since these members won't be
used).
    * The given function is called implicitly by the runtime when the property is set
    * from JavaScript code (or if a set on the property is performed using a N-API call).
    */
    .setter = NULL,
/*
    * The value that's retrieved by a get access of the property if the property is a
data property.
    * If this is passed in, set getter, setter, method and data to NULL (since these
members won't be used).
    */
    .value = NULL,
/*
    * The attributes associated with the particular property. See
napi_property_attributes.
    */
    .attributes = napi_default,
/*
    * The callback data passed into method, getter and setter if this function is
invoked.
    */
    .data = NULL
};
/*
    * This method allows the efficient definition of multiple properties on a given object.
    */
status = napi_define_properties(env, exports, 1, &desc);

if (status != napi_ok)
    return;
}

NAPI_MODULE(hello, init)

```

Lea N-API en línea: <https://riptutorial.com/es/node-js/topic/10539/n-api>

Capítulo 82: Node.js (express.js) con código de ejemplo angular.js

Introducción

Este ejemplo muestra cómo crear una aplicación Express básica y luego servir a AngularJS.

Examples

Creando nuestro proyecto.

Estamos bien para ir así, corremos, de nuevo desde la consola:

```
mkdir our_project
cd our_project
```

Ahora estamos en el lugar donde vivirá nuestro código. Para crear el archivo principal de nuestro proyecto puede ejecutar

Ok, pero ¿cómo creamos el proyecto del esqueleto expreso?

Es sencillo:

```
npm install -g express express-generator
```

Las distribuciones de Linux y Mac deben usar **sudo** para instalar esto porque están instaladas en el directorio nodejs, al que solo accede el usuario **root** . Si todo salió bien, podemos, finalmente, crear el esqueleto de la aplicación Express, simplemente ejecutar

```
express
```

Este comando creará dentro de nuestra carpeta una aplicación de ejemplo express. La estructura es la siguiente:

```
bin/
public/
routes/
views/
app.js
package.json
```

Ahora, si ejecutamos **npm, inicie** y vaya a <http://localhost:3000> veremos la aplicación Express en funcionamiento, lo suficiente, hemos generado una aplicación Express sin demasiados problemas, pero ¿cómo podemos mezclar esto con AngularJS? .

¿Cómo expreso funciona, brevemente?

Express es un marco construido sobre **Nodejs** , puede ver la documentación oficial en el [sitio Express](#) . Pero para nuestro propósito, necesitamos saber que **Express** es el responsable cuando escribimos, por ejemplo, [http: // localhost: 3000 / home](http://localhost:3000/home) al renderizar la página de inicio de nuestra aplicación. Desde la aplicación creada recientemente podemos verificar:

```
FILE: routes/index.js
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

Lo que nos dice este código es que cuando el usuario accede a [http: // localhost: 3000](http://localhost:3000) , debe mostrar la vista de **índice** y pasar un **JSON** con una propiedad de título y un valor Express. Pero cuando revisamos el directorio de vistas y abrimos index.jade, podemos ver esto:

```
extends layout
block content
  h1= title
  p Welcome to #{title}
```

Esta es otra característica poderosa de Express, los **motores de plantillas** , que le permiten representar contenido en la página pasándole variables o heredar otra plantilla para que sus páginas sean más compactas y más comprensibles para los demás. La extensión del archivo es **.jade** , que yo sepa, **Jade** cambió el nombre de **Pug** , básicamente es el mismo motor de plantillas pero con algunas actualizaciones y modificaciones principales.

Instalando Pug y actualizando el motor de plantillas Express.

Ok, para comenzar a usar Pug como motor de plantillas de nuestro proyecto, necesitamos ejecutar:

```
npm install --save pug
```

Esto instalará Pug como una dependencia de nuestro proyecto y lo guardará en **package.json** . Para usarlo necesitamos modificar el archivo **app.js** :

```
var app = express();
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

Y reemplazar el motor de línea de vista con pug y eso es todo. Podemos volver a ejecutar nuestro proyecto con **npm start** y veremos que todo está funcionando bien.

¿Cómo encaja AngularJS en todo esto?

AngularJS es un **marco de Javascript MVW** (Model-View-Whatever) utilizado principalmente para crear **SPA** (aplicación de página simple). La instalación es bastante simple, puede ir al [sitio web de AngularJS](#) y descargar la última versión **v1.6.4**.

Después de descargar AngularJS cuando deberíamos copiar el archivo a nuestra carpeta **pública / javascripts** dentro de nuestro proyecto, una pequeña explicación, esta es la carpeta que sirve a los recursos estáticos de nuestro sitio, imágenes, css, archivos javascript, etc. Por supuesto, esto es configurable a través del archivo **app.js**, pero lo mantendremos simple. Ahora creamos un archivo llamado **ng-app.js**, el archivo donde vivirá nuestra aplicación, dentro de nuestra carpeta pública javascripts, justo donde vive AngularJS. Para abrir AngularJS necesitamos modificar el contenido de **views / layout.pug** de la siguiente manera:

```
doctype html
html (ng-app='first-app')
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body (ng-controller='indexController')
    block content

    script (type='text-javascript', src='javascripts/angular.min.js')
    script (type='text-javascript', src='javascripts/ng-app.js')
```

¿Qué estamos haciendo aquí? Bueno, estamos incluyendo el núcleo de AngularJS y nuestro archivo creado recientemente **ng-app.js**, de modo que cuando se **muestre** la plantilla, aparecerá AngularJS. Observe el uso de la directiva **ng-app**. AngularJS dice que este es nuestro nombre de aplicación y que debe atenerse a él.

Entonces, el contenido de nuestro **ng-app.js** será:

```
angular.module('first-app', [])
  .controller('indexController', ['$scope', indexController]);

function indexController($scope) {
  $scope.name = 'sigfried';
}
```

Estamos utilizando la función más básica de AngularJS aquí, **enlace de datos bidireccional**, esto nos permite actualizar el contenido de nuestra vista y el controlador al instante, esta es una explicación muy simple, pero puede hacer una investigación en Google o StackOverflow para ver como realmente funciona

Entonces, tenemos los bloques básicos de nuestra aplicación AngularJS, pero hay algo que tenemos que hacer, necesitamos actualizar nuestra página index.pug para ver los cambios de nuestra aplicación angular, hagámoslo:

```
extends layout
block content
  div (ng-controller='indexController')
    h1= title
    p Welcome {{name}}
    input (type='text' ng-model='name')
```

Aquí solo estamos vinculando la entrada a nuestro nombre de propiedad definido en el ámbito de AngularJS dentro de nuestro controlador:

```
$scope.name = 'sigfried';
```

El propósito de esto es que siempre que cambiemos el texto en la entrada, el párrafo anterior actualizará el contenido dentro de {{nombre}}, esto se llama **interpolación**, otra característica de AngularJS para representar nuestro contenido en la plantilla.

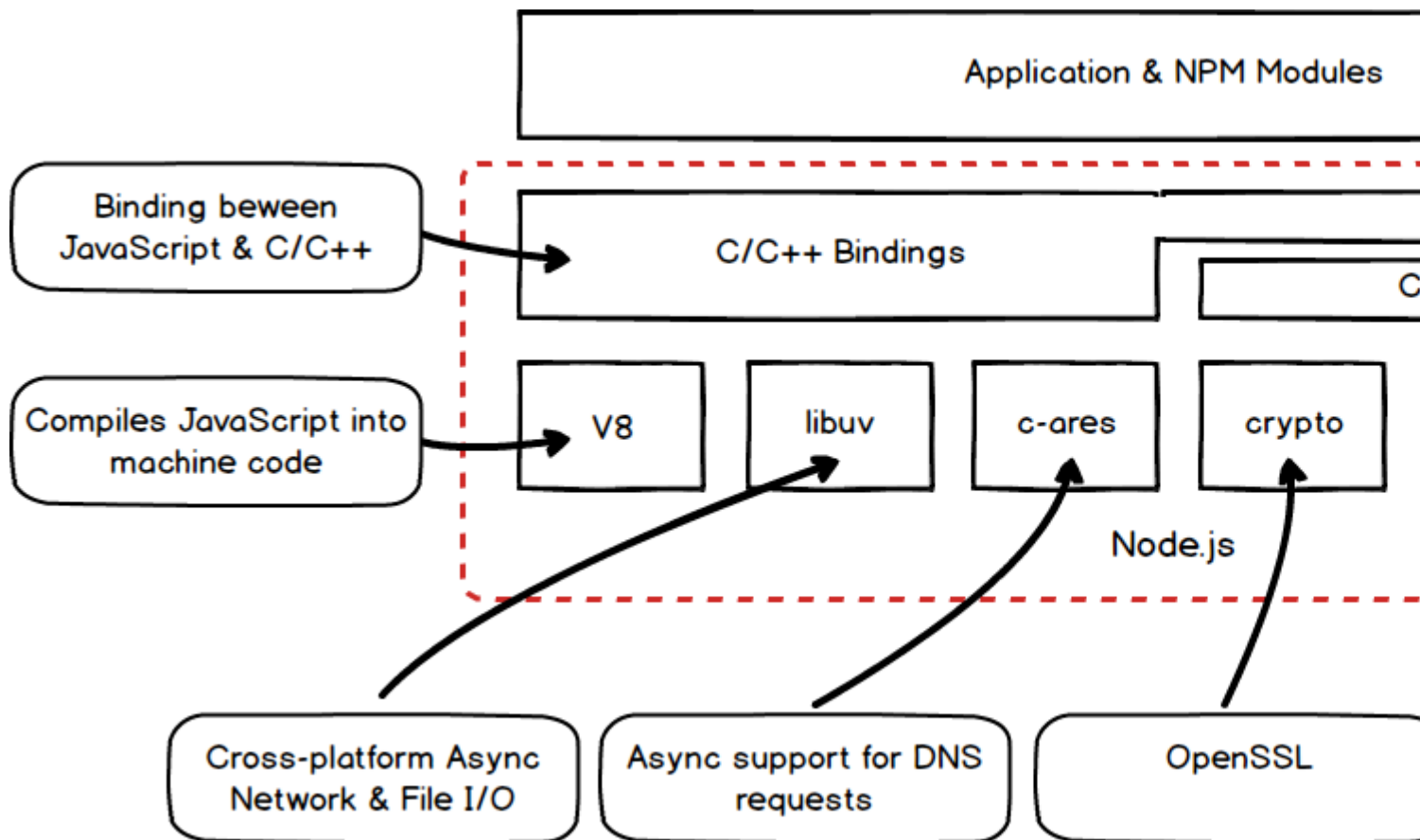
Entonces, todo está configurado, ahora podemos ejecutar **npm start**, vaya a <http://localhost:3000> y vea nuestra aplicación Express al servicio de la página y AngularJS administrando la interfaz de la aplicación.

Lea [Node.js \(express.js\) con código de ejemplo angular.js en línea](https://riptutorial.com/es/node-js/topic/9757/node-js--express-js--con-codigo-de-ejemplo-angular-js): <https://riptutorial.com/es/node-js/topic/9757/node-js--express-js--con-codigo-de-ejemplo-angular-js>

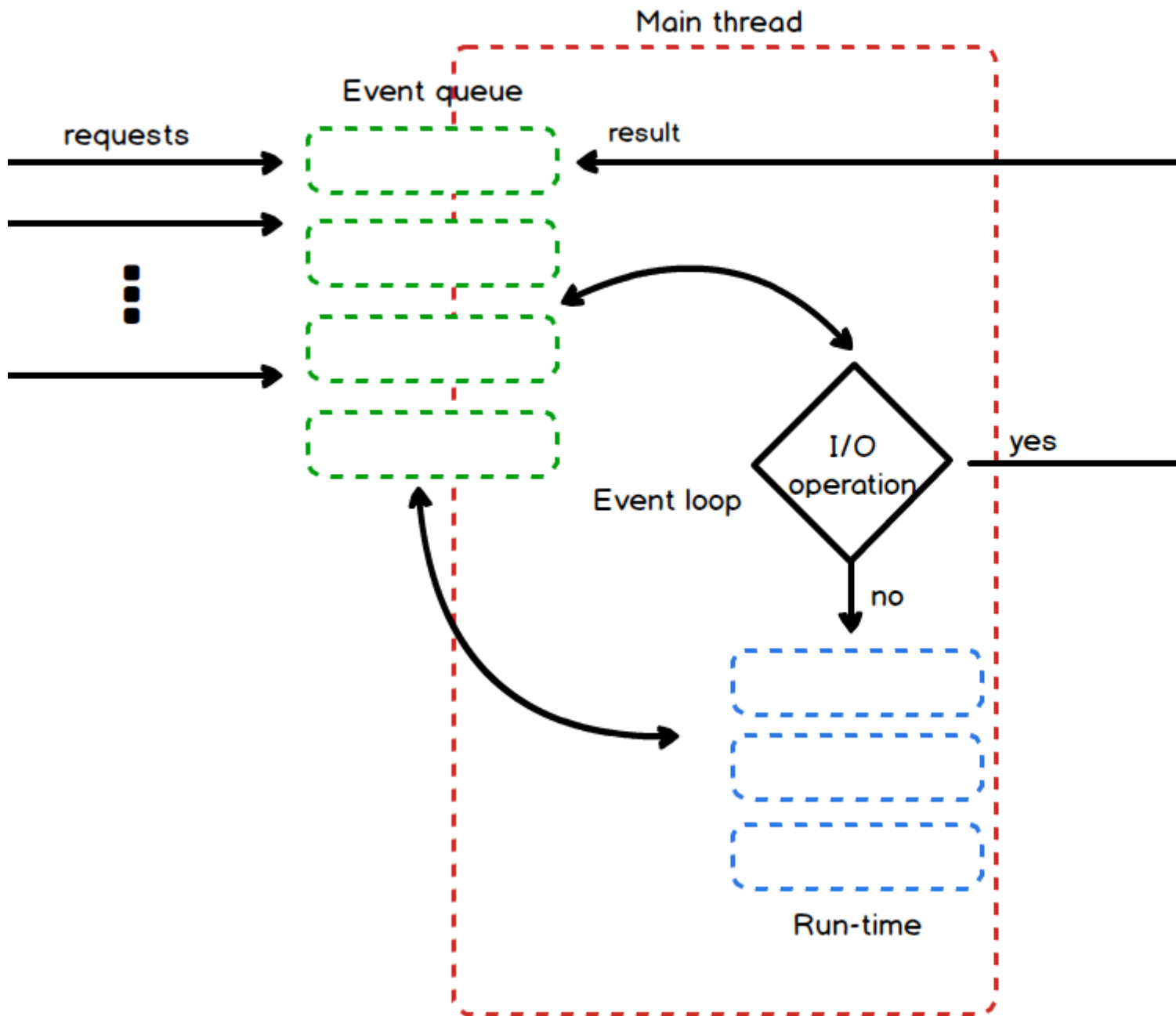
Capítulo 83: Node.js Arquitectura y Trabajos Internos

Examples

Node.js - bajo el capó



Node.js - en movimiento



Lea Node.js Arquitectura y Trabajos Internos en línea: <https://riptutorial.com/es/node-js/topic/5892/node-js-arquitectura-y-trabajos-internos>

Capítulo 84: Node.js con CORS

Examples

Habilitar CORS en express.js

Como node.js se usa a menudo para crear API, la configuración CORS adecuada puede ser un salvavidas si desea poder solicitar la API desde diferentes dominios.

En el ejemplo, lo configuraremos para una configuración más amplia (autorizaremos todos los tipos de solicitud desde cualquier dominio).

En su server.js después de inicializar express:

```
// Create express server
const app = express();

app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', '*');

  // authorized headers for preflight requests
  // https://developer.mozilla.org/en-US/docs/Glossary/preflight_request
  res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type,
Accept');
  next();

  app.options('*', (req, res) => {
    // allowed XHR methods
    res.header('Access-Control-Allow-Methods', 'GET, PATCH, PUT, POST, DELETE, OPTIONS');
    res.send();
  });
});
```

Generalmente, el nodo se ejecuta detrás de un proxy en servidores de producción. Por lo tanto, el servidor proxy inverso (como Apache o Nginx) será responsable de la configuración de CORS.

Para adaptar convenientemente este escenario, es posible habilitar solo el nodo.js CORS cuando está en desarrollo.

Esto se hace fácilmente marcando `NODE_ENV` :

```
const app = express();

if (process.env.NODE_ENV === 'development') {
  // CORS settings
}
```

Lea Node.js con CORS en línea: <https://riptutorial.com/es/node-js/topic/9272/node-js-con-cors>

Capítulo 85: Node.JS con ES6

Introducción

ES6, ECMAScript 6 o ES2015 es la última [especificación](#) para JavaScript que introduce algo de azúcar sintáctico al lenguaje. Es una gran actualización del lenguaje e introduce muchas [características](#) nuevas.

Se pueden encontrar más detalles sobre Node y ES6 en su sitio <https://nodejs.org/en/docs/es6/>

Examples

Nodo ES6 Soporte y creación de un proyecto con Babel.

La especificación completa de ES6 aún no se ha implementado en su totalidad, por lo que solo podrá usar algunas de las nuevas funciones. Puede ver una lista de las funciones ES6 compatibles actuales en <http://node.green/>

Desde NodeJS v6 ha habido bastante buen soporte. Por lo tanto, si usa NodeJS v6 o superior, puede disfrutar de usar ES6. Sin embargo, es posible que también desee utilizar algunas de las características inéditas y otras del más allá. Para ello necesitarás utilizar un transpiler.

Es posible ejecutar un transpiler en tiempo de ejecución y compilación, para usar todas las características de ES6 y más. El transpiler más popular para JavaScript se llama [Babel](#)

Babel le permite usar todas las características de la especificación ES6 y algunas características adicionales no-en-especificación con 'stage-0' como `import thing from 'thing'` lugar de `var thing = require('thing')`

Si quisiéramos crear un proyecto en el que usáramos las características de 'stage-0', como la importación, tendríamos que agregar Babel como un transpiler. Verá que los proyectos que usan React y Vue y otros patrones comunes basados en JS implementan la etapa 0 con bastante frecuencia.

crear un nuevo proyecto de nodo

```
mkdir my-es6-app
cd my-es6-app
npm init
```

Instala babel el preset ES6 y stage-0

```
npm install --save-dev babel-preset-es2015 babel-preset-stage-2 babel-cli babel-register
```

Cree un nuevo archivo llamado `server.js` y agregue un servidor HTTP básico.

```
import http from 'http'
```

```
http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'})
  res.end('Hello World\n')
}).listen(3000, '127.0.0.1')

console.log('Server running at http://127.0.0.1:3000/')
```

Tenga en cuenta que usamos un `import http from 'http'` esta es una función de etapa 0 y, si funciona, significa que el transpiler funciona correctamente.

Si ejecuta `node server.js`, fallará al no saber cómo manejar la importación.

Creando un archivo `.babelrc` en la raíz de su directorio y agregue la siguiente configuración

```
{
  "presets": ["es2015", "stage-2"],
  "plugins": []
}
```

ahora puede ejecutar el servidor con el `node src/index.js --exec babel-node`

Para terminar, no es una buena idea ejecutar un transpiler en tiempo de ejecución en una aplicación de producción. Sin embargo, podemos implementar algunos scripts en nuestro `package.json` para que sea más fácil trabajar con ellos.

```
"scripts": {
  "start": "node dist/index.js",
  "dev": "babel-node src/index.js",
  "build": "babel src -d dist",
  "postinstall": "npm run build"
},
```

Lo anterior en `npm install` construirá el código transpilado en el directorio `dist`, permitiendo que `npm start` a utilizar el código transpilado para nuestra aplicación de producción.

`npm run dev` iniciará el servidor y el tiempo de ejecución de babel, lo cual es correcto y se prefiere cuando se trabaja en un proyecto localmente.

A continuación, puede instalar nodemon `npm install nodemon --save-dev` para observar los cambios y luego reiniciar la aplicación del nodo.

Esto realmente acelera el trabajo con babel y NodeJS. En tu `package.json` solo actualiza el script `"dev"` para usar nodemon

```
"dev": "nodemon src/index.js --exec babel-node",
```

Usa JS es6 en tu aplicación NodeJS

JS es6 (también conocido como es2015) es un conjunto de nuevas características para el lenguaje JS cuyo objetivo es hacerlo más intuitivo al usar OOP o al enfrentar tareas de desarrollo modernas.