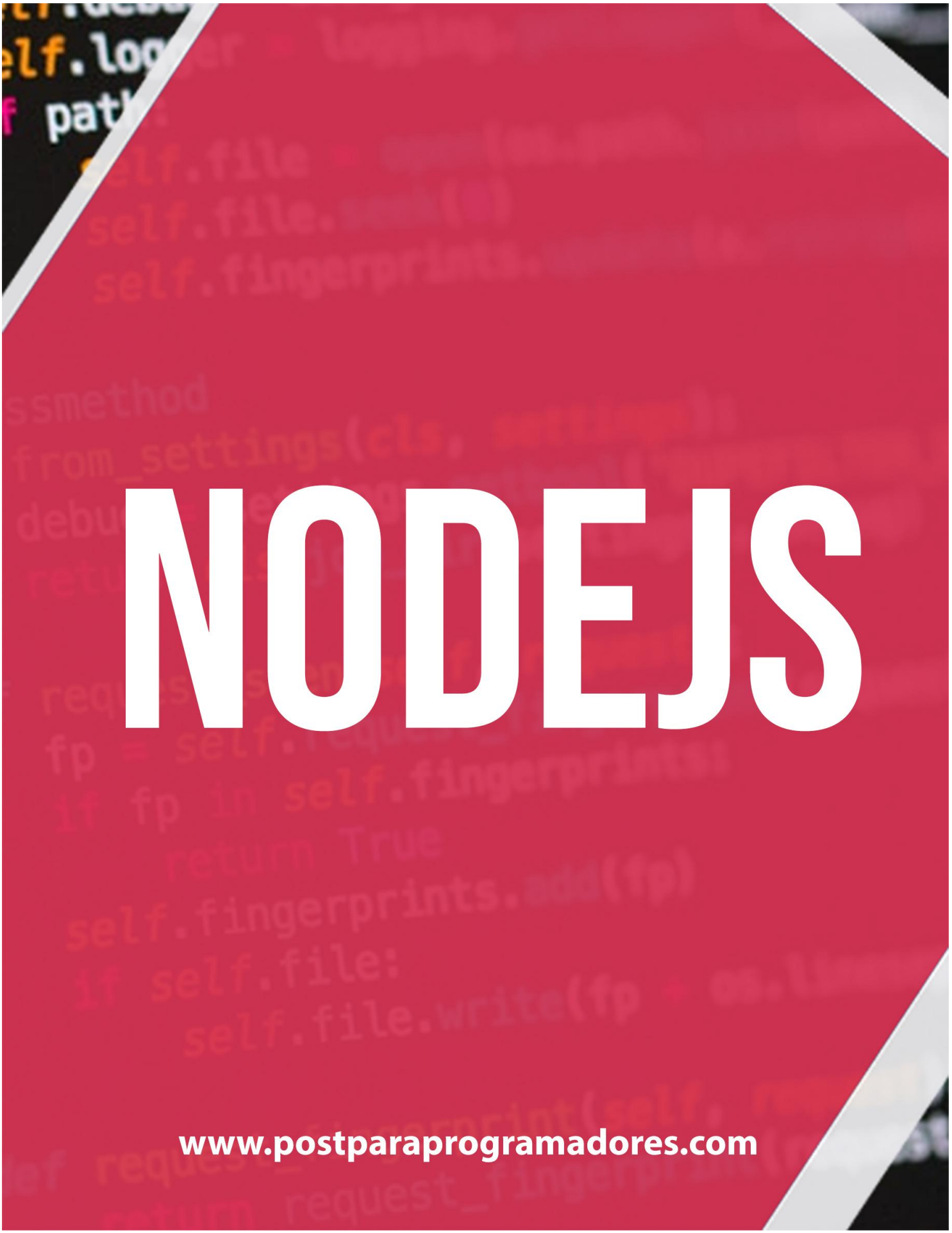

NODEJS
www.postparaprogramadores.com

Contenido

Node.js - Inicio

Node.js - Introducción

Node.js - Configuración del entorno

Node.js - Primera aplicación

Node.js - Terminal REPL

Node.js - Administrador de paquetes (NPM)

Node.js - Concepto de devoluciones de llamada

Node.js - Bucle de eventos

Node.js - Emisor de eventos

Node.js - Buffers

Node.js - Streams

Node.js - Sistema de archivos

Node.js - Objetos globales

Node.js - Módulos de utilidad

Node.js - Módulo web

Node.js - Express Framework

Node.js - API RESTFul

Node.js - Aplicación de escalado

Node.js - Embalaje

Node.js - Introducción

¿Qué es Node.js?

Node.js es una plataforma del lado del servidor construida en el motor JavaScript de Google Chrome (motor V8). Node.js fue desarrollado por Ryan Dahl en 2009 y su última versión es v0.10.36. La definición de Node.js tal como la proporciona su documentación oficial es la siguiente:

> Node.js es una plataforma basada en el tiempo de ejecución de JavaScript de Chrome para crear fácilmente aplicaciones de red rápidas y escalables. Node.js utiliza un modelo de E / S sin bloqueo controlado por eventos que lo hace liviano y eficiente, perfecto para aplicaciones en tiempo real de uso intensivo de datos que se ejecutan en dispositivos distribuidos.

Node.js es un entorno de tiempo de ejecución multiplataforma de código abierto para desarrollar aplicaciones de red y del lado del servidor. Las aplicaciones Node.js están escritas en JavaScript y pueden ejecutarse dentro del tiempo de ejecución de Node.js en OS X, Microsoft Windows y Linux.

Node.js también proporciona una rica biblioteca de varios módulos JavaScript que simplifica el desarrollo de aplicaciones web usando Node.js en gran medida.

```
Node.js = Runtime Environment + JavaScript Library
```

Características de Node.js

Las siguientes son algunas de las características importantes que hacen de Node.js la primera opción de arquitectos de software.

- **Asíncrono y controlado por eventos** : todas las API de la biblioteca Node.js son asíncronas, es decir, sin bloqueo. Esencialmente significa que un servidor basado en Node.js nunca espera que una API devuelva datos. El servidor pasa a la siguiente API después de llamarlo y un mecanismo de notificación de Events of Node.js ayuda al servidor a obtener una respuesta de la llamada API anterior.
- **Muy rápido** : al estar construido en el motor JavaScript V8 de Google Chrome, la biblioteca Node.js es muy rápida en la ejecución de código.
- **Rosca única pero altamente escalable** : Node.js utiliza un modelo de una sola rosca con bucle de eventos. El mecanismo de eventos ayuda al servidor a responder sin bloqueos y hace que el servidor sea altamente escalable en comparación con los servidores tradicionales que crean hilos limitados para manejar las solicitudes. Node.js utiliza un solo programa de subprocesos y el mismo programa puede proporcionar servicio a un número mucho mayor de solicitudes que los servidores tradicionales como el Servidor Apache HTTP.
- **Sin** almacenamiento en búfer: las aplicaciones Node.js nunca almacenan en búfer ningún dato. Estas aplicaciones simplemente generan los datos en fragmentos.
- **Licencia** : Node.js se publica bajo la licencia MIT .

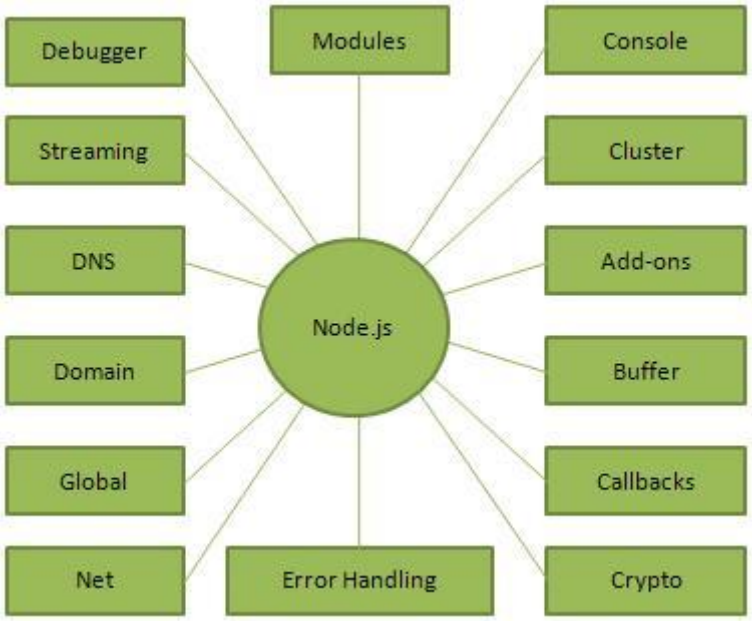
¿Quién usa Node.js?

El siguiente es el enlace en github wiki que contiene una lista exhaustiva de proyectos, aplicaciones y empresas que están utilizando Node.js. Esta lista incluye eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo !, y Yammer, por nombrar algunos.

- Proyectos, aplicaciones y empresas que utilizan el nodo

Conceptos

El siguiente diagrama muestra algunas partes importantes de Node.js que discutiremos en detalle en los capítulos siguientes.



¿Dónde usar Node.js?

Las siguientes son las áreas donde Node.js se está demostrando como un socio tecnológico perfecto.

- Aplicaciones vinculadas de E / S
- Aplicaciones de transmisión de datos
- Aplicaciones intensivas de datos en tiempo real (DIRT)
- Aplicaciones basadas en API JSON
- Aplicaciones de una sola página

¿Dónde no usar Node.js?

No es aconsejable usar Node.js para aplicaciones intensivas de CPU.

Node.js - Configuración del entorno

Pruébelo Opción en línea

Realmente no necesita configurar su propio entorno para comenzar a aprender Node.js. La razón es muy simple, ya hemos configurado el entorno Node.js en línea, para que pueda ejecutar todos los ejemplos disponibles en línea y aprender a través de la práctica. No dude en modificar cualquier ejemplo y verifique los resultados con diferentes opciones.

Pruebe el siguiente ejemplo con la opción de **demostración en vivo** disponible en la esquina superior derecha del cuadro de código de muestra siguiente (en nuestro sitio web):

Demo en vivo

```
/* Hello World! program in Node.js */
console.log("Hello World!");
```

Para la mayoría de los ejemplos dados en este tutorial, encontrará una opción Pruébelo, así que solo utilícelo y disfrute de su aprendizaje.

Configuración del entorno local

Si todavía está dispuesto a configurar su entorno para Node.js, necesita los siguientes dos softwares disponibles en su computadora, (a) Editor de texto y (b) Los archivos binarios instalables de Node.js.

Editor de texto

Esto se usará para escribir su programa. Los ejemplos de algunos editores incluyen el Bloc de notas de Windows, el comando Editar del sistema operativo, Breve, Epsilon, EMACS y vim o vi.

El nombre y la versión del editor de texto pueden variar en diferentes sistemas operativos. Por ejemplo, el Bloc de notas se usará en Windows y vim o vi se pueden usar en Windows, así como en Linux o UNIX.

Los archivos que crea con su editor se denominan archivos fuente y contienen el código fuente del programa. Los archivos de origen para los programas Node.js generalmente se nombran con la extensión " **.js** ".

Antes de comenzar su programación, asegúrese de tener un editor de texto y tener suficiente experiencia para escribir un programa de computadora, guardarlo en un archivo y finalmente ejecutarlo.

El tiempo de ejecución de Node.js

El código fuente escrito en el archivo fuente es simplemente javascript. El intérprete Node.js se usará para interpretar y ejecutar su código javascript.

La distribución Node.js viene como un binario instalable para sistemas operativos SunOS, Linux, Mac OS X y Windows con arquitecturas de procesador x86 de 32 bits (386) y 64 bits (amd64).

La siguiente sección lo guía sobre cómo instalar la distribución binaria Node.js en varios sistemas operativos.

Descargar el archivo Node.js

Descarga la última versión del fichero de archivo instalable de Node.js Node.js Descargas . Al momento de escribir este tutorial, a continuación se encuentran las versiones disponibles en diferentes sistemas operativos.

OS	Nombre de archivo
Ventanas	nodo-v6.3.1-x64.msi

Linux	node-v6.3.1-linux-x86.tar.gz
Mac	node-v6.3.1-darwin-x86.tar.gz
SunOS	node-v6.3.1-sunos-x86.tar.gz

Instalación en UNIX / Linux / Mac OS X y SunOS

Según la arquitectura de su sistema operativo, descargue y extraiga el archivo archive node-v6.3.1- **osname** .tar.gz en / tmp, y finalmente mueva los archivos extraídos al directorio / usr / local / nodejs. Por ejemplo:

```
$ cd /tmp
$ wget http://nodejs.org/dist/v6.3.1/node-v6.3.1-linux-
x64.tar.gz
$ tar xvfz node-v6.3.1-linux-x64.tar.gz
$ mkdir -p /usr/local/nodejs
$ mv node-v6.3.1-linux-x64/* /usr/local/nodejs
```

Agregue / usr / local / nodejs / bin a la variable de entorno PATH.

OS	Salida
Linux	RUTA de exportación = $ RUTA: / usr / local / nodejs / bin
Mac	RUTA de exportación = $ RUTA: / usr / local / nodejs / bin
FreeBSD	RUTA de exportación = $ RUTA: / usr / local / nodejs / bin

Instalación en Windows

Use el archivo MSI y siga las instrucciones para instalar Node.js. De manera predeterminada, el instalador usa la distribución Node.js en C: \ Archivos de programa \ nodejs. El instalador debe establecer el directorio C: \ Archivos de programa \ nodejs \ bin en la variable de entorno PATH de la ventana. Reinicie cualquier solicitud de comando abierto para que el cambio surta efecto.

Verificar instalación: Ejecutar un archivo

Cree un archivo js llamado **main.js** en su máquina (Windows o Linux) que tenga el siguiente código.

```
/* Hello, World! program in node.js */
console.log("Hello, World!")
```

Ahora ejecute el archivo main.js con el intérprete Node.js para ver el resultado:

```
$ node main.js
```

Si todo está bien con su instalación, esto debería producir el siguiente resultado:

```
Hello, World!
```

Node.js - Primera aplicación

Antes de crear un verdadero "¡Hola, mundo!" aplicación usando Node.js, veamos los componentes de una aplicación Node.js. Una aplicación Node.js consta de los siguientes tres componentes importantes:

- **Importar módulos requeridos** : utilizamos la directiva **require** para cargar módulos Node.js.
- **Crear servidor** : un servidor que escuchará las solicitudes del cliente similares a Apache HTTP Server.
- **Leer solicitud y respuesta de respuesta** : el servidor creado en un paso anterior leerá la solicitud HTTP realizada por el cliente, que puede ser un navegador o una consola, y devolverá la respuesta.

Crear la aplicación Node.js

Paso 1 - Importar módulo requerido

Usamos la directiva **require** para cargar el módulo http y almacenar la instancia HTTP devuelta en una variable http de la siguiente manera:

```
var http = require("http");
```

Paso 2 - Crear servidor

Usamos la instancia http creada y llamamos al método **http.createServer ()** para crear una instancia de servidor y luego la **vinculamos** en el puerto 8081 usando el método de **escucha** asociado con la instancia del servidor. Pásalo una función con parámetros de solicitud y respuesta. Escriba la implementación de muestra para devolver siempre "Hello World".

```
http.createServer(function (request, response) {
   // Send the HTTP header
   // HTTP Status: 200 : OK
   // Content Type: text/plain
   response.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body as "Hello World"
   response.end('Hello World\n');
}).listen(8081);
```

```
// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

El código anterior es suficiente para crear un servidor HTTP que escucha, es decir, espera una solicitud sobre el puerto 8081 en la máquina local.

Paso 3 - Solicitud de prueba y respuesta

Pongamos los pasos 1 y 2 juntos en un archivo llamado **main.js** e **iniciemos** nuestro servidor HTTP como se muestra a continuación:

```
var http = require("http");

http.createServer(function (request, response) {
   // Send the HTTP header
   // HTTP Status: 200 : OK
   // Content Type: text/plain
   response.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body as "Hello World"
   response.end('Hello World\n');
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

Ahora ejecute main.js para iniciar el servidor de la siguiente manera:

```
$ node main.js
```

Verifique la salida. El servidor ha comenzado.

```
Server running at http://127.0.0.1:8081/
```

Hacer una solicitud al servidor Node.js

Abra http://127.0.0.1:8081/ en cualquier navegador y observe el siguiente resultado.

Felicidades, tiene su primer servidor HTTP en funcionamiento que responde a todas las solicitudes HTTP en el puerto 8081.

Node.js - Terminal REPL

REPL significa Read Eval Print Loop y representa un entorno informático como una consola de Windows o un shell de Unix / Linux donde se ingresa un comando y el sistema responde con una salida en un modo interactivo. Node.js o **Node** viene incluido con un entorno REPL. Realiza las siguientes tareas:

- **Leer** : lee la entrada del usuario, analiza la entrada en la estructura de datos de JavaScript y la almacena en la memoria.
- **Eval** - Toma y evalúa la estructura de datos.
- **Imprimir** : imprime el resultado.
- **Bucle** : repite el comando anterior hasta que el usuario presione **ctrl-c** dos veces.

La función REPL de Node es muy útil para experimentar con los códigos Node.js y para depurar códigos JavaScript.

Terminal REPL en línea

Para simplificar su aprendizaje, hemos configurado un entorno REPL Node.js fácil de usar en línea, donde puede practicar la sintaxis de Node.js - Inicie la Terminal REPL Node.js

Iniciando REPL

REPL puede iniciarse simplemente ejecutando el **nodo** en la consola / consola sin ningún argumento de la siguiente manera.

```
$ node
```

Verá el símbolo del sistema REPL> donde puede escribir cualquier comando Node.js:

```
$ node
>
```

Expresión simple

Probemos una matemática simple en el símbolo del sistema Node.js REPL:

```
$ node
> 1 + 3
4
> 1 + ( 2 * 3 ) - 4
3
>
```

Usar variables

Puede hacer uso de variables para almacenar valores e imprimir más tarde como cualquier script convencional. Si no se usa la palabra clave **var** , el valor se almacena en la variable y se imprime. Mientras que si se usa la palabra clave **var** , el valor se almacena pero no se imprime. Puede imprimir variables usando **console.log ()** .

```
$ node
> x = 10
10
> var y = 10
undefined
> x + y
20
> console.log("Hello World")
Hello World
undefined
```

Expresión Multilínea

El nodo REPL admite expresiones multilíneas similares a JavaScript. Veamos el siguiente ciclo do-while en acción:

```
$ node
> var x = 0
undefined
> do {
   ... x++;
   ... console.log("x: " + x);
   ... }
while ( x < 5 );
x: 1
x: 2
x: 3
x: 4
```

```
x: 5
undefined
>
```

... viene automáticamente cuando presiona Enter después del corchete de apertura. El nodo verifica automáticamente la continuidad de las expresiones.

Subrayar variable

Puede usar el guión bajo **(_)** para obtener el último resultado:

```
$ node
> var x = 10
undefined
> var y = 20
undefined
> x + y
30
> var sum = _
undefined
> console.log(sum)
30
undefined
>
```

Comandos REPL

- **ctrl + c** - termina el comando actual.
- **ctrl + c dos veces** - termina el Nodo REPL.
- **ctrl + d** - termina el Nodo REPL.
- **Teclas arriba / abajo** : vea el historial de comandos y modifique los comandos anteriores.
- **Teclas de tabulación** : lista de comandos actuales.
- **.help** : lista de todos los comandos.
- **.break** : sale de la expresión multilínea.
- **.clear** : sale de la expresión multilínea.
- **.save *filename*** : guarda la sesión actual de Node REPL en un archivo.
- **.load *filename*** : carga el contenido del archivo en la sesión REPL Nodo actual.

Deteniendo REPL

Como se mencionó anteriormente, deberá usar **ctrl-c dos veces** para salir de Node.js REPL.

```
$ node
>
(^C again to quit)
>
```

Node.js - NPM

Node Package Manager (NPM) proporciona dos funcionalidades principales:

- Repositorios en línea para paquetes / módulos node.js que se pueden buscar en search.nodejs.org
- Utilidad de línea de comandos para instalar paquetes Node.js, hacer gestión de versiones y gestión de dependencias de paquetes Node.js.

NPM viene incluido con los instalables de Node.js después de la versión v0.6.3. Para verificar lo mismo, abra la consola y escriba el siguiente comando y vea el resultado:

```
$ npm --version
2.7.1
```

Si está ejecutando una versión anterior de NPM, es bastante fácil actualizarla a la última versión. Simplemente use el siguiente comando desde la raíz:

```
$ sudo npm install npm -g
/usr/bin/npm -> /usr/lib/node_modules/npm/bin/npm-cli.js
npm@2.7.1 /usr/lib/node_modules/npm
```

Instalación de módulos usando NPM

Hay una sintaxis simple para instalar cualquier módulo Node.js:

```
$ npm install <Module Name>
```

Por ejemplo, el siguiente es el comando para instalar un famoso módulo de marco web Node.js llamado express -

```
$ npm install express
```

Ahora puede usar este módulo en su archivo js de la siguiente manera:

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

Instalación global versus local

Por defecto, NPM instala cualquier dependencia en el modo local. Aquí el modo local se refiere a la instalación del paquete en el directorio node_modules que se encuentra en la carpeta donde está presente la aplicación Node. Se puede acceder a los paquetes implementados localmente mediante el método require (). Por ejemplo, cuando instalamos el módulo express, creó el directorio node_modules en el directorio actual donde instaló el módulo express.

```
$ ls -l
total 0
drwxr-xr-x 3 root root 20 Mar 17 02:23 node_modules
```

Alternativamente, puede usar el **comando npm ls** para enumerar todos los módulos instalados localmente.

Los paquetes / dependencias instalados globalmente se almacenan en el directorio del sistema. Dichas dependencias se pueden utilizar en la función CLI (interfaz de línea de comandos) de cualquier nodo.js, pero no se pueden importar utilizando require () en la aplicación Node directamente. Ahora intentemos instalar el módulo express utilizando la instalación global.

```
$ npm install express -g
```

Esto producirá un resultado similar, pero el módulo se instalará globalmente. Aquí, la primera línea muestra la versión del módulo y la ubicación donde se está instalando.

```
express@4.12.2 /usr/lib/node_modules/express
├── merge-descriptors@1.0.0
├── utils-merge@1.0.0
├── cookie-signature@1.0.6
├── methods@1.1.1
├── fresh@0.2.4
├── cookie@0.1.2
├── escape-html@1.0.1
├── range-parser@1.0.2
├── content-type@1.0.1
├── finalhandler@0.3.3
├── vary@1.0.0
├── parseurl@1.3.0
├── content-disposition@0.5.0
├── path-to-regexp@0.1.3
├── depd@1.0.0
├── qs@2.3.3
├── on-finished@2.2.0 (ee-first@1.1.0)
├── etag@1.5.1 (crc@3.2.1)
├── debug@2.1.3 (ms@0.7.0)
├── proxy-addr@1.0.7 (forwarded@0.1.0, ipaddr.js@0.1.9)
├── send@0.12.1 (destroy@1.0.3, ms@0.7.0, mime@1.3.4)
├── serve-static@1.9.2 (send@0.12.2)
├── accepts@1.2.5 (negotiator@0.5.1, mime-types@2.0.10)
└── type-is@1.6.1 (media-typer@0.3.0, mime-types@2.0.10)
```

Puede usar el siguiente comando para verificar todos los módulos instalados globalmente:

```
$ npm ls -g
```

Usando package.json

package.json está presente en el directorio raíz de cualquier aplicación / módulo Node y se usa para definir las propiedades de un paquete. Abramos package.json del paquete express presente en **node_modules / express /**

```
{
   "name": "express",
      "description": "Fast, unopinionated, minimalist web
framework",
      "version": "4.11.2",
```

```
    "author": {

       "name": "TJ Holowaychuk",
       "email": "tj@vision-media.ca"
    },

 "contributors": [{
    "name": "Aaron Heckmann",
    "email": "aaron.heckmann+github@gmail.com"
 },

 {
    "name": "Ciaran Jessup",
    "email": "ciaranj@gmail.com"
 },

 {
    "name": "Douglas Christopher Wilson",
    "email": "doug@somethingdoug.com"
 },

 {
    "name": "Guillermo Rauch",
    "email": "rauchg@gmail.com"
 },

 {
    "name": "Jonathan Ong",
    "email": "me@jongleberry.com"
 },

 {
    "name": "Roman Shtylman",
    "email": "shtylman+expressjs@gmail.com"
 },

 {
    "name": "Young Jae Sim",
    "email": "hanul@hanul.me"
 } ],

 "license": "MIT", "repository": {
    "type": "git",
    "url": "https://github.com/strongloop/express"
 },

 "homepage": "https://expressjs.com/", "keywords": [
    "express",
    "framework",
    "sinatra",
    "web",
    "rest",
    "restful",
```

```
    "router",
    "app",
    "api"
  ],

  "dependencies": {
    "accepts": "~1.2.3",
    "content-disposition": "0.5.0",
    "cookie-signature": "1.0.5",
    "debug": "~2.1.1",
    "depd": "~1.0.0",
    "escape-html": "1.0.1",
    "etag": "~1.5.1",
    "finalhandler": "0.3.3",
    "fresh": "0.2.4",
    "media-typer": "0.3.0",
    "methods": "~1.1.1",
    "on-finished": "~2.2.0",
    "parseurl": "~1.3.0",
    "path-to-regexp": "0.1.3",
    "proxy-addr": "~1.0.6",
    "qs": "2.3.3",
    "range-parser": "~1.0.2",
    "send": "0.11.1",
    "serve-static": "~1.8.1",
    "type-is": "~1.5.6",
    "vary": "~1.0.0",
    "cookie": "0.1.2",
    "merge-descriptors": "0.0.2",
    "utils-merge": "1.0.0"
  },

  "devDependencies": {
    "after": "0.8.1",
    "ejs": "2.1.4",
    "istanbul": "0.3.5",
    "marked": "0.3.3",
    "mocha": "~2.1.0",
    "should": "~4.6.2",
    "supertest": "~0.15.0",
    "hjs": "~0.0.6",
    "body-parser": "~1.11.0",
    "connect-redis": "~2.2.0",
    "cookie-parser": "~1.3.3",
    "express-session": "~1.10.2",
    "jade": "~1.9.1",
    "method-override": "~2.3.1",
    "morgan": "~1.5.1",
    "multiparty": "~4.1.1",
    "vhost": "~3.0.0"
  },

  "engines": {
```

```
      "node": ">= 0.10.0"
   },

   "files": [
      "LICENSE",
      "History.md",
      "Readme.md",
      "index.js",
      "lib/"
   ],

   "scripts": {
      "test": "mocha --require test/support/env
         --reporter spec --bail --check-leaks test/
test/acceptance/",
      "test-cov": "istanbul cover
node_modules/mocha/bin/_mocha
         -- --require test/support/env --reporter dot --
check-leaks test/ test/acceptance/",
      "test-tap": "mocha --require test/support/env
         --reporter tap --check-leaks test/
test/acceptance/",
      "test-travis": "istanbul cover
node_modules/mocha/bin/_mocha
         --report lcovonly -- --require test/support/env
         --reporter spec --check-leaks test/
test/acceptance/"
   },

   "gitHead": "63ab25579bda70b4927a179b580a9c580b6c7ada",
   "bugs": {
      "url": "https://github.com/strongloop/express/issues"
   },

   "_id": "express@4.11.2",
   "_shasum": "8df3d5a9ac848585f00a0777601823faecd3b148",
   "_from": "express@*",
   "_npmVersion": "1.4.28",
   "_npmUser": {
      "name": "dougwilson",
      "email": "doug@somethingdoug.com"
   },

   "maintainers": [{
      "name": "tjholowaychuk",
      "email": "tj@vision-media.ca"
   },

   {
      "name": "jongleberry",
      "email": "jonathanrichardong@gmail.com"
   },
```

```
    {
       "name": "shtylman",
       "email": "shtylman@gmail.com"
    },

    {
       "name": "dougwilson",
       "email": "doug@somethingdoug.com"
    },

    {
       "name": "aredridel",
       "email": "aredridel@nbtsc.org"
    },

    {
       "name": "strongloop",
       "email": "callback@strongloop.com"
    },

    {
       "name": "rfeng",
       "email": "enjoyjava@gmail.com"
    }],

    "dist": {
       "shasum": "8df3d5a9ac848585f00a0777601823faecd3b148",
       "tarball": "https://registry.npmjs.org/express/-
/express-4.11.2.tgz"
    },

    "directories": {},
       "_resolved": "https://registry.npmjs.org/express/-
/express-4.11.2.tgz",
       "readme": "ERROR: No README data found!"
}
```

Atributos de Package.json

- **nombre** - nombre del paquete
- **version** - versión del paquete
- **description** - descripción del paquete
- **página de inicio** - página de inicio del paquete
- **autor** - autor del paquete
- **contribuyentes** - nombre de los contribuyentes al paquete
- **dependencias** : lista de dependencias. NPM instala automáticamente todas las dependencias mencionadas aquí en la carpeta node_module del paquete.
- **repositorio** - tipo de repositorio y URL del paquete

- **main** - punto de entrada del paquete
- **palabras clave** - palabras clave

Desinstalar un módulo

Use el siguiente comando para desinstalar un módulo Node.js.

```
$ npm uninstall express
```

Una vez que NPM desinstala el paquete, puede verificarlo mirando el contenido del directorio / node_modules / o escriba el siguiente comando:

```
$ npm ls
```

Actualización de un módulo

Actualice package.json y cambie la versión de la dependencia que se actualizará y ejecute el siguiente comando.

```
$ npm update express
```

Buscar un módulo

Busque el nombre de un paquete usando NPM.

```
$ npm search express
```

Crear un módulo

La creación de un módulo requiere que se genere package.json. Generemos package.json usando NPM, lo que generará el esqueleto básico de package.json.

```
$ npm init
This utility will walk you through creating a package.json
file.
It only covers the most common items, and tries to guess sane
defaults.

See 'npm help json' for definitive documentation on these
fields
and exactly what they do.

Use 'npm install <pkg> --save' afterwards to install a
package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (webmaster)
```

Deberá proporcionar toda la información requerida sobre su módulo. Puede obtener ayuda del archivo package.json mencionado anteriormente para comprender los significados de la información solicitada. Una vez que se

genera package.json, use el siguiente comando para registrarse en el sitio del repositorio de NPM utilizando una dirección de correo electrónico válida.

```
$ npm adduser
Username: mcmohd
Password:
Email: (this IS public) mcmohd@gmail.com
```

Es hora de publicar su módulo.

```
$ npm publish
```

Si todo está bien con su módulo, se publicará en el repositorio y será accesible para instalar usando NPM como cualquier otro módulo Node.js.

Node.js - Concepto de devoluciones de llamada

¿Qué es la devolución de llamada?

La devolución de llamada es un equivalente asíncrono para una función. Se llama a una función de devolución de llamada al finalizar una tarea determinada. Node hace un uso intensivo de las devoluciones de llamada. Todas las API de Node están escritas de tal manera que admiten devoluciones de llamada.

Por ejemplo, una función para leer un archivo puede comenzar a leer el archivo y devolver el control al entorno de ejecución inmediatamente para que se pueda ejecutar la siguiente instrucción. Una vez que se completa la E / S del archivo, llamará a la función de devolución de llamada mientras pasa la función de devolución de llamada, el contenido del archivo como parámetro. Por lo tanto, no hay bloqueo ni espere la E / S de archivo. Esto hace que Node.js sea altamente escalable, ya que puede procesar una gran cantidad de solicitudes sin esperar que ninguna función devuelva resultados.

Ejemplo de código de bloqueo

Cree un archivo de texto llamado **input.txt** con el siguiente contenido:

```
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
```

Cree un archivo js llamado **main.js** con el siguiente código:

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("Program Ended");
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
Program Ended
```

Ejemplo de código sin bloqueo

Cree un archivo de texto llamado input.txt con el siguiente contenido.

```
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
```

Actualice main.js para que tenga el siguiente código:

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
   if (err) return console.error(err);
   console.log(data.toString());
});

console.log("Program Ended");
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Program Ended
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
```

Estos dos ejemplos explican el concepto de llamadas bloqueadas y no bloqueantes.

- El primer ejemplo muestra que el programa se bloquea hasta que lee el archivo y luego solo se procede a finalizar el programa.
- El segundo ejemplo muestra que el programa no espera la lectura del archivo y procede a imprimir "Programa finalizado" y, al mismo tiempo, el programa sin bloqueo continúa leyendo el archivo.

Por lo tanto, un programa de bloqueo se ejecuta mucho en secuencia. Desde el punto de vista de la programación, es más fácil implementar la lógica, pero los programas sin bloqueo no se ejecutan en secuencia. En caso de que un programa necesite usar cualquier dato para ser procesado, debe mantenerse dentro del mismo bloque para que sea una ejecución secuencial.

Node.js - Bucle de eventos

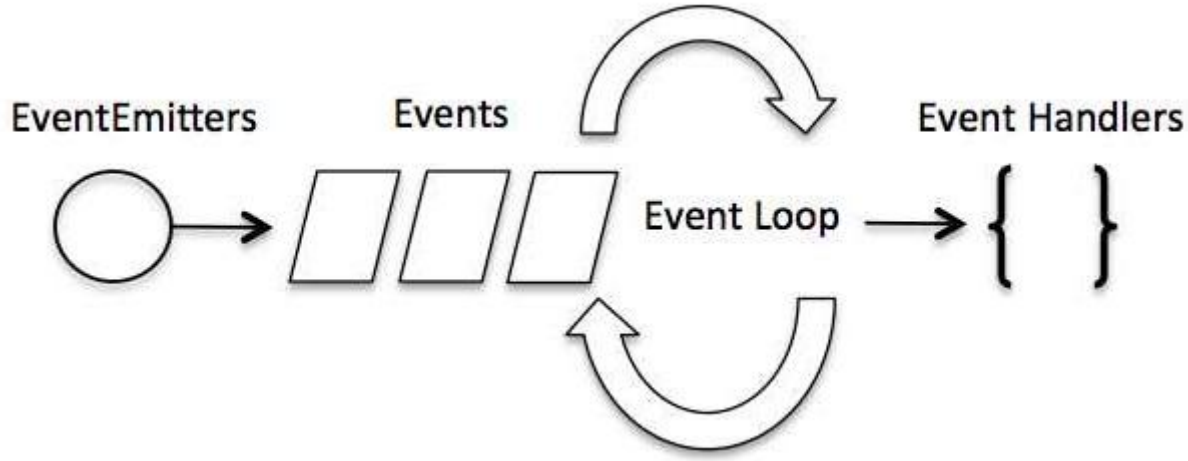
Node.js es una aplicación de subproceso único, pero puede admitir concurrencia a través del concepto de **evento** y **devoluciones de llamada** . Cada API de Node.js es asíncrona y tiene un solo subproceso, utilizan **llamadas de función asíncronas** para mantener la concurrencia. El nodo utiliza el patrón de observador. El subproceso de nodo mantiene un

bucle de eventos y cada vez que se completa una tarea, dispara el evento correspondiente que indica que se ejecute la función de escucha de eventos.

Programación dirigida por eventos

Node.js utiliza los eventos en gran medida y también es una de las razones por las que Node.js es bastante rápido en comparación con otras tecnologías similares. Tan pronto como Node inicia su servidor, simplemente inicia sus variables, declara funciones y luego simplemente espera que ocurra el evento.

En una aplicación controlada por eventos, generalmente hay un bucle principal que escucha los eventos y luego activa una función de devolución de llamada cuando se detecta uno de esos eventos.



Aunque los eventos se parecen bastante a las devoluciones de llamada, la diferencia radica en el hecho de que las funciones de devolución de llamada se invocan cuando una función asincrónica devuelve su resultado, mientras que el manejo de eventos funciona en el patrón del observador. Las funciones que escuchan eventos actúan como **observadores** . Cada vez que se dispara un evento, su función de escucha comienza a ejecutarse. Node.js tiene varios eventos incorporados disponibles a través del módulo de eventos y la clase EventEmitter que se utilizan para vincular eventos y oyentes de eventos de la siguiente manera:

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
```

La siguiente es la sintaxis para vincular un controlador de eventos con un evento:

```
// Bind event and event  handler as follows
eventEmitter.on('eventName', eventHandler);
```

Podemos activar un evento mediante programación de la siguiente manera:

```
// Fire an event
eventEmitter.emit('eventName');
```

Ejemplo

Cree un archivo js llamado main.js con el siguiente código:

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();

// Create an event handler as follows
var connectHandler = function connected() {
   console.log('connection succesful.');

   // Fire the data_received event
   eventEmitter.emit('data_received');
}

// Bind the connection event with the handler
eventEmitter.on('connection', connectHandler);

// Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function() {
   console.log('data received succesfully.');
});

// Fire the connection event
eventEmitter.emit('connection');

console.log("Program Ended.");
```

Ahora intentemos ejecutar el programa anterior y verificar su salida:

```
$ node main.js
```

TI debería producir el siguiente resultado:

```
connection successful.
data received successfully.
Program Ended.
```

¿Cómo funcionan las aplicaciones de nodo?

En la aplicación de nodo, cualquier función asíncrona acepta una devolución de llamada como último parámetro y una función de devolución de llamada acepta un error como primer parámetro. Volvamos al ejemplo anterior nuevamente. Cree un archivo de texto llamado input.txt con el siguiente contenido.

```
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
```

Cree un archivo js llamado main.js que tenga el siguiente código:

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
   if (err) {
      console.log(err.stack);
      return;
   }
   console.log(data.toString());
});
console.log("Program Ended");
```

Aquí fs.readFile () es una función asíncrona cuyo propósito es leer un archivo. Si se produce un error durante la operación de lectura, el **objeto err** contendrá el error correspondiente; de lo contrario, los datos contendrán el contenido del archivo. **readFile** pasa err y datos a la función de devolución de llamada después de que se completa la operación de lectura, que finalmente imprime el contenido.

```
Program Ended
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
```

Node.js - Emisor de eventos

Muchos objetos en un nodo emiten eventos, por ejemplo, una red. El servidor emite un evento cada vez que un par se conecta a él, un fs.readStream emite un evento cuando se abre el archivo. Todos los objetos que emiten eventos son instancias de eventos.EventEmitter.

Clase EventEmitter

Como hemos visto en la sección anterior, la clase EventEmitter se encuentra en el módulo de eventos. Se puede acceder a través del siguiente código:

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
```

Cuando una instancia de EventEmitter se enfrenta a un error, emite un evento de 'error'. Cuando se agrega un nuevo oyente, se activa el evento 'newListener' y cuando se elimina un oyente, se activa el evento 'removeListener'.

EventEmitter proporciona múltiples propiedades como **encendido** y **emisión** . **La** propiedad **on** se usa para vincular una función con el evento y **emit** se usa para disparar un evento.

Métodos

No Señor.	Método y descripción
1	**addListener (evento, oyente)** Agrega un oyente al final de la matriz de oyentes para el evento especificado. No se realizan verificaciones para ver si el oyente ya se ha agregado. Las llamadas múltiples que pasan la misma combinación de evento y escucha harán que el escucha se agregue varias veces. Devuelve el emisor, por lo que las llamadas se pueden encadenar.
2	**encendido (evento, oyente)** Agrega un oyente al final de la matriz de oyentes para el evento especificado. No se realizan verificaciones para ver si el oyente ya se ha agregado. Las llamadas múltiples que pasan la misma combinación de evento y escucha harán que el escucha se agregue varias veces. Devuelve el emisor, por lo que las llamadas se pueden encadenar.
3	**una vez (evento, oyente)** Agrega un oyente único al evento. Este oyente se invoca solo la próxima vez que se dispara el evento, después de lo cual se elimina. Devuelve el emisor, por lo que las llamadas se pueden encadenar.
4 4	**removeListener (evento, oyente)** Elimina un oyente de la matriz de oyentes para el evento especificado. **Precaución:** cambia los índices de la matriz en la matriz del oyente detrás del oyente. removeListener eliminará, como máximo, una instancia de un oyente de la matriz de oyentes. Si se ha agregado varias veces un único oyente a la matriz de oyentes para el evento especificado, se debe llamar a removeListener varias veces para eliminar cada instancia. Devuelve el emisor, por lo que las llamadas se pueden encadenar.
5 5	**removeAllListeners ([evento])** Elimina todos los oyentes, o los del evento especificado. No es una buena idea eliminar los oyentes que se agregaron en otra parte del código, especialmente cuando está en un emisor que no creó (por ejemplo, sockets o secuencias de archivos). Devuelve el emisor, por lo que las llamadas se pueden encadenar.
6 6	**setMaxListeners (n)** Por defecto, EventEmitters imprimirá una advertencia si se agregan más de 10 oyentes para un evento en particular. Este es un valor predeterminado útil que ayuda a encontrar pérdidas de memoria. Obviamente, no todos los Emisores deberían limitarse a 10. Esta función permite que se incremente. Establecer en cero para ilimitado.

7 7	**oyentes (evento)** Devuelve una matriz de oyentes para el evento especificado.
8	**emitir (evento, [arg1], [arg2], [...])** Ejecute cada uno de los oyentes en orden con los argumentos proporcionados. Devuelve verdadero si el evento tuvo oyentes, falso de lo contrario.

Métodos de clase

No Señor.	Método y descripción
1	**listenerCount (emisor, evento)** Devuelve el número de oyentes para un evento determinado.

Eventos

No Señor.	Eventos y descripción
1	**newListener** • **evento** - Cadena: el nombre del evento • **oyente** - Función: la función de controlador de eventos Este evento se emite cada vez que se agrega un oyente. Cuando se desencadena este evento, es posible que el oyente aún no se haya agregado a la matriz de oyentes para el evento.
2	**removeListener** • **event** - String El nombre del evento • **oyente** - Función La función de controlador de eventos Este evento se emite cada vez que alguien elimina a un oyente. Cuando se desencadena este evento, es posible que el oyente aún no se haya eliminado de la matriz de oyentes para el evento.

Ejemplo

Cree un archivo js llamado main.js con el siguiente código Node.js:

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

// listener #1
var listner1 = function listner1() {
   console.log('listner1 executed.');
}

// listener #2
var listner2 = function listner2() {
   console.log('listner2 executed.');
}

// Bind the connection event with the listner1 function
eventEmitter.addListener('connection', listner1);

// Bind the connection event with the listner2 function
eventEmitter.on('connection', listner2);

var eventListeners =
require('events').EventEmitter.listenerCount
   (eventEmitter,'connection');
console.log(eventListeners + " Listner(s) listening to
connection event");

// Fire the connection event
eventEmitter.emit('connection');

// Remove the binding of listner1 function
eventEmitter.removeListener('connection', listner1);
console.log("Listner1 will not listen now.");

// Fire the connection event
eventEmitter.emit('connection');

eventListeners =
require('events').EventEmitter.listenerCount(eventEmitter,'co
nnection');
console.log(eventListeners + " Listner(s) listening to
connection event");

console.log("Program Ended.");
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
2 Listner(s) listening to connection event
listner1 executed.
listner2 executed.
```

```
Listner1 will not listen now.
listner2 executed.
1 Listner(s) listening to connection event
Program Ended.
```

Node.js - Buffers

JavaScript puro es compatible con Unicode, pero no es así para los datos binarios. Al tratar con flujos TCP o el sistema de archivos, es necesario manejar flujos octetos. Node proporciona la clase Buffer que proporciona instancias para almacenar datos sin formato similares a una matriz de enteros, pero corresponde a una asignación de memoria sin formato fuera del montón V8.

La clase de búfer es una clase global a la que se puede acceder desde una aplicación sin importar el módulo de búfer.

Creando Buffers

Node Buffer se puede construir de varias maneras.

Método 1

La siguiente es la sintaxis para crear un búfer no iniciado de **10** octetos:

```
var buf = new Buffer(10);
```

Método 2

La siguiente es la sintaxis para crear un Buffer a partir de una matriz dada:

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

Método 3

La siguiente es la sintaxis para crear un Buffer a partir de una cadena dada y, opcionalmente, un tipo de codificación:

```
var buf = new Buffer("Simply Easy Learning", "utf-8");
```

Aunque "utf8" es la codificación predeterminada, puede usar cualquiera de las siguientes codificaciones "ascii", "utf8", "utf16le", "ucs2", "base64" o "hex".

Escribiendo a Buffers

Sintaxis

La siguiente es la sintaxis del método para escribir en un Node Buffer:

```
buf.write(string[, offset][, length][, encoding])
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **string** : estos son los datos de cadena que se escribirán en el búfer.
- **offset** : este es el índice del búfer en el que comenzar a escribir. El valor predeterminado es 0.
- **longitud** : este es el número de bytes a escribir. Por defecto es buffer.length.
- **codificación** : codificación para usar. 'utf8' es la codificación predeterminada.

Valor de retorno

Este método devuelve el número de octetos escritos. Si no hay suficiente espacio en el búfer para caber toda la cadena, escribirá una parte de la cadena.

Ejemplo

```
buf = new Buffer(256);
len = buf.write("Simply Easy Learning");

console.log("Octets written : "+  len);
```

Cuando se ejecuta el programa anterior, produce el siguiente resultado:

```
Octets written : 20
```

Leyendo de Buffers

Sintaxis

La siguiente es la sintaxis del método para leer datos de un Node Buffer:

```
buf.toString([encoding][, start][, end])
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **codificación** : codificación para usar. 'utf8' es la codificación predeterminada.
- **inicio** : índice inicial para comenzar a leer, el valor predeterminado es 0.
- **end** : el índice final para finalizar la lectura, el valor predeterminado es el búfer completo

Valor de retorno

Este método decodifica y devuelve una cadena de datos de búfer codificados utilizando la codificación del juego de caracteres especificado.

Ejemplo

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}

console.log( buf.toString('ascii'));       // outputs:
abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));   // outputs: abcde
console.log( buf.toString('utf8',0,5));    // outputs: abcde
console.log( buf.toString(undefined,0,5)); // encoding
defaults to 'utf8', outputs abcde
```

Cuando se ejecuta el programa anterior, produce el siguiente resultado:

```
abcdefghijklmnopqrstuvwxyz
abcde
abcde
abcde
```

Convertir Buffer a JSON

Sintaxis

La siguiente es la sintaxis del método para convertir un Node Buffer en un objeto JSON:

```
buf.toJSON()
```

Valor de retorno

Este método devuelve una representación JSON de la instancia de Buffer.

Ejemplo

```
var buf = new Buffer('Simply Easy Learning');
var json = buf.toJSON(buf);

console.log(json);
```

Cuando se ejecuta el programa anterior, produce el siguiente resultado:

```
{ type: 'Buffer',
   data:
   [
      83,
      105,
      109,
      112,
      108,
      121,
      32,
```

```
      69,
      97,
      115,
      121,
      32,
      76,
      101,
      97,
      114,
      110,
      105,
      110,
      103
   ]
}
```

Tampones Concatenados

Sintaxis

A continuación se muestra la sintaxis del método para concatenar las memorias intermedias de nodo a una única memoria intermedia de nodo:

```
Buffer.concat(list[, totalLength])
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **list** - Lista de matriz de objetos Buffer que se concatenarán.
- **totalLength** : esta es la longitud total de los búferes cuando se concatenan.

Valor de retorno

Este método devuelve una instancia de Buffer.

Ejemplo

```
var buffer1 = new Buffer('Postparaprogramadores ');
var buffer2 = new Buffer('Simply Easy Learning');
var buffer3 = Buffer.concat([buffer1,buffer2]);

console.log("buffer3 content: " + buffer3.toString());
```

Cuando se ejecuta el programa anterior, produce el siguiente resultado:

```
buffer3 content: Postparaprogramadores Simply Easy Learning
```

Comparar tampones

Sintaxis

A continuación se muestra la sintaxis del método para comparar dos buffers de nodo:

```
buf.compare(otherBuffer);
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **otherBuffer** : este es el otro búfer que se comparará con **buf**

Valor de retorno

Devuelve un número que indica si viene antes o después o si es el mismo que el otro Buffer en orden de clasificación.

Ejemplo

```
var buffer1 = new Buffer('ABC');
var buffer2 = new Buffer('ABCD');
var result = buffer1.compare(buffer2);

if(result < 0) {
   console.log(buffer1 +" comes before " + buffer2);
} else if(result === 0) {
   console.log(buffer1 +" is same as " + buffer2);
} else {
   console.log(buffer1 +" comes after " + buffer2);
}
```

Cuando se ejecuta el programa anterior, produce el siguiente resultado:

```
ABC comes before ABCD
```

Copia de búfer

Sintaxis

A continuación se muestra la sintaxis del método para copiar un búfer de nodo:

```
buf.copy(targetBuffer[, targetStart][, sourceStart][,
sourceEnd])
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **targetBuffer** : objeto de almacenamiento intermedio donde se copiará el almacenamiento intermedio.

- **targetStart** - Número, Opcional, Predeterminado: 0
- **sourceStart** - Number, Opcional, Predeterminado: 0
- **sourceEnd** - Number, Opcional, Predeterminado: buffer.length

Valor de retorno

Sin valor de retorno. Copia datos de una región de este búfer a una región en el búfer de destino, incluso si la región de memoria de destino se superpone con la fuente. Si no está definido, los parámetros targetStart y sourceStart tienen un valor predeterminado de 0, mientras que sourceEnd tiene el valor predeterminado de buffer.length.

Ejemplo

```
var buffer1 = new Buffer('ABC');

//copy a buffer
var buffer2 = new Buffer(3);
buffer1.copy(buffer2);
console.log("buffer2 content: " + buffer2.toString());
```

Cuando se ejecuta el programa anterior, produce el siguiente resultado:

```
buffer2 content: ABC
```

Tampón de rebanada

Sintaxis

A continuación se muestra la sintaxis del método para obtener un sub-búfer de un búfer de nodo:

```
buf.slice([start][, end])
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **inicio** - Número, Opcional, Predeterminado: 0
- **end** - Número, Opcional, Predeterminado: buffer.length

Valor de retorno

Devuelve un nuevo búfer que hace referencia a la misma memoria que el anterior, pero compensado y recortado por los índices de inicio (predeterminado a 0) y final (predeterminado a buffer.length). Los índices negativos comienzan desde el final del búfer.

Ejemplo

```
var buffer1 = new Buffer('Postparaprogramadores');

//slicing a buffer
var buffer2 = buffer1.slice(0,9);
console.log("buffer2 content: " + buffer2.toString());
```

Cuando se ejecuta el programa anterior, produce el siguiente resultado:

```
buffer2 content: Tutorials
```

Longitud del almacenador intermediario

Sintaxis

A continuación se muestra la sintaxis del método para obtener el tamaño de un búfer de nodo en bytes:

```
buf.length;
```

Valor de retorno

Devuelve el tamaño de un búfer en bytes.

Ejemplo

```
var buffer = new Buffer('Postparaprogramadores');

//length of the buffer
console.log("buffer length: " + buffer.length);
```

Cuando se ejecuta el programa anterior, produce el siguiente resultado:

```
buffer length: 14
```

Referencia de métodos

A continuación se incluye una referencia del módulo Buffers disponible en Node.js. Para más detalles, puede consultar la documentación oficial.

Métodos de clase

No Señor.	Método y descripción
1	**Buffer.isEncoding (codificación)** Devuelve verdadero si la codificación es un argumento de codificación válido, falso

	en caso contrario.
2	**Buffer.isBuffer (obj)** Comprueba si obj es un Buffer.
3	**Buffer.byteLength (cadena [, codificación])** Da la longitud de bytes real de una cadena. la codificación por defecto es 'utf8'. No es lo mismo que String.prototype.length, ya que String.prototype.length devuelve el número de caracteres en una cadena.
4 4	**Buffer.concat (list [, totalLength])** Devuelve un búfer que es el resultado de concatenar todos los búferes de la lista juntos.
5 5	**Buffer.compare (buf1, buf2)** Lo mismo que buf1.compare (buf2). Útil para ordenar una variedad de buffers.

Node.js - Streams

¿Qué son las corrientes?

Las secuencias son objetos que le permiten leer datos de una fuente o escribir datos en un destino de manera continua. En Node.js, hay cuatro tipos de transmisiones:

- **Legible** : secuencia que se utiliza para la operación de lectura.
- **Writable** : secuencia que se utiliza para la operación de escritura.
- **Dúplex** : flujo que se puede utilizar para operaciones de lectura y escritura.
- **Transformar** : un tipo de flujo dúplex donde la salida se calcula en función de la entrada.

Cada tipo de Stream es una instancia de **EventEmitter** y lanza varios eventos en diferentes instancias de tiempos. Por ejemplo, algunos de los eventos más utilizados son:

- **datos** : este evento se activa cuando hay datos disponibles para leer.
- **end** : este evento se activa cuando no hay más datos para leer.
- **error** : este evento se activa cuando hay algún error al recibir o escribir datos.
- **terminar** : este evento se activa cuando todos los datos se han vaciado al sistema subyacente.

Este tutorial proporciona una comprensión básica de las operaciones comúnmente utilizadas en Streams.

Leer desde una corriente

Cree un archivo de texto llamado input.txt que tenga el siguiente contenido:

```
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
```

Cree un archivo js llamado main.js con el siguiente código:

```
var fs = require("fs");
var data = '';

// Create a readable stream
var readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});

readerStream.on('end',function() {
   console.log(data);
});

readerStream.on('error', function(err) {
   console.log(err.stack);
});

console.log("Program Ended");
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Program Ended
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
```

Escribir en una secuencia

Cree un archivo js llamado main.js con el siguiente código:

```
var fs = require("fs");
var data = 'Simply Easy Learning';

// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');

// Write the data to stream with encoding to be utf8
```

```
writerStream.write(data,'UTF8');

// Mark the end of file
writerStream.end();

// Handle stream events --> finish, and error
writerStream.on('finish', function() {
   console.log("Write completed.");
});

writerStream.on('error', function(err) {
   console.log(err.stack);
});

console.log("Program Ended");
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Program Ended
Write completed.
```

Ahora abra output.txt creado en su directorio actual; debe contener lo siguiente:

```
Simply Easy Learning
```

Canalizando las corrientes

La tubería es un mecanismo en el que proporcionamos la salida de un flujo como la entrada a otro flujo. Normalmente se usa para obtener datos de una secuencia y para pasar la salida de esa secuencia a otra secuencia. No hay límite en las operaciones de tuberías. Ahora mostraremos un ejemplo de tubería para leer de un archivo y escribirlo en otro archivo.

Cree un archivo js llamado main.js con el siguiente código:

```
var fs = require("fs");

// Create a readable stream
var readerStream = fs.createReadStream('input.txt');

// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');

// Pipe the read and write operations
// read input.txt and write data to output.txt
readerStream.pipe(writerStream);

console.log("Program Ended");
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Program Ended
```

Abra output.txt creado en su directorio actual; debe contener lo siguiente:

```
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
```

Encadenando las corrientes

El encadenamiento es un mecanismo para conectar la salida de un flujo a otro flujo y crear una cadena de operaciones de flujo múltiple. Normalmente se usa con operaciones de tuberías. Ahora usaremos tuberías y cadenas para comprimir primero un archivo y luego descomprimirlo.

Cree un archivo js llamado main.js con el siguiente código:

```
var fs = require("fs");
var zlib = require('zlib');

// Compress the file input.txt to input.txt.gz
fs.createReadStream('input.txt')
   .pipe(zlib.createGzip())
   .pipe(fs.createWriteStream('input.txt.gz'));

console.log("File Compressed.");
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
File Compressed.
```

Encontrará que input.txt ha sido comprimido y creó un archivo input.txt.gz en el directorio actual. Ahora intentemos descomprimir el mismo archivo usando el siguiente código:

```
var fs = require("fs");
var zlib = require('zlib');

// Decompress the file input.txt.gz to input.txt
fs.createReadStream('input.txt.gz')
   .pipe(zlib.createGunzip())
   .pipe(fs.createWriteStream('input.txt'));

console.log("File Decompressed.");
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
File Decompressed.
```

Node.js - Sistema de archivos

Node implementa File I / O usando envoltorios simples alrededor de funciones POSIX estándar. El módulo Sistema de archivos de nodo (fs) se puede importar utilizando la siguiente sintaxis:

```
var fs = require("fs")
```

Sincrónico vs Asincrónico

Cada método en el módulo fs tiene formas síncronas y asíncronas. Los métodos asincrónicos toman el último parámetro como la devolución de llamada de la función de finalización y el primer parámetro de la función de devolución de llamada como error. Es mejor usar un método asincrónico en lugar de un método sincrónico, ya que el primero nunca bloquea un programa durante su ejecución, mientras que el segundo sí.

Ejemplo

Cree un archivo de texto llamado **input.txt** con el siguiente contenido:

```
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
```

Creemos un archivo js llamado **main.js** con el siguiente código:

```
var fs = require("fs");

// Asynchronous read
fs.readFile('input.txt', function (err, data) {
   if (err) {
      return console.error(err);
   }
   console.log("Asynchronous read: " + data.toString());
});

// Synchronous read
var data = fs.readFileSync('input.txt');
console.log("Synchronous read: " + data.toString());

console.log("Program Ended");
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Synchronous read: Postparaprogramadores is giving self
learning content
to teach the world in simple and easy way!!!!!

Program Ended
Asynchronous read: Postparaprogramadores is giving self
learning content
```

```
to teach the world in simple and easy way!!!!!
```

Las siguientes secciones de este capítulo proporcionan un conjunto de buenos ejemplos sobre los principales métodos de E / S de archivos.

Abrir un archivo

Sintaxis

La siguiente es la sintaxis del método para abrir un archivo en modo asíncrono:

```
fs.open(path, flags[, mode], callback)
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **ruta** : esta es la cadena que tiene el nombre del archivo, incluida la ruta.
- **flags** : los flags indican el comportamiento del archivo a abrir. Todos los valores posibles se han mencionado a continuación.
- **modo** : establece el modo de archivo (permiso y bits fijos), pero solo si se creó el archivo. El valor predeterminado es 0666, legible y grabable.
- **devolución de llamada** : esta es la función de devolución de llamada que obtiene dos argumentos (err, fd).

Banderas

Las banderas para las operaciones de lectura / escritura son:

No Señor.	Bandera y descripción
1	**r** Abrir archivo para leer. Se produce una excepción si el archivo no existe.
2	**r +** Abrir archivo para leer y escribir. Se produce una excepción si el archivo no existe.
3	**rs** Abrir archivo para leer en modo síncrono.
4 4	**rs +** Abra el archivo para leer y escribir, pidiéndole al sistema operativo que lo abra

	sincrónicamente. Consulte las notas para 'rs' sobre el uso de esto con precaución.
5 5	**w** Abrir archivo para escribir. El archivo se crea (si no existe) o se trunca (si existe).
6 6	**wx** Como 'w' pero falla si la ruta existe.
7 7	**w +** Abrir archivo para leer y escribir. El archivo se crea (si no existe) o se trunca (si existe).
8	**wx +** Como 'w +' pero falla si la ruta existe.
9 9	**un** Abrir archivo para anexar. El archivo se crea si no existe.
10	**hacha** Como 'a' pero falla si la ruta existe.
11	**a +** Abrir archivo para leer y agregar. El archivo se crea si no existe.
12	**hacha +** Como 'a +' pero falla si la ruta existe.

Ejemplo

Creemos un archivo js llamado **main.js que** tenga el siguiente código para abrir un archivo input.txt para leer y escribir.

```
var fs = require("fs");

// Asynchronous - Opening File
console.log("Going to open file!");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
      return console.error(err);
   }
```

```
   console.log("File opened successfully!");
});
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Going to open file!
File opened successfully!
```

Obtener información de archivo

Sintaxis

La siguiente es la sintaxis del método para obtener la información sobre un archivo:

```
fs.stat(path, callback)
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **ruta** : esta es la cadena que tiene el nombre del archivo, incluida la ruta.
- **devolución de llamada** : esta es la función de devolución de llamada que obtiene dos argumentos (err, stats) donde **stats** es un objeto de tipo fs.Stats que se imprime a continuación en el ejemplo.

Además de los atributos importantes que se imprimen a continuación en el ejemplo, hay varios métodos útiles disponibles en la clase **fs.Stats** que se pueden usar para verificar el tipo de archivo. Estos métodos se dan en la siguiente tabla.

No Señor.	Método y descripción
1	**stats.isFile ()** Devuelve verdadero si el tipo de archivo de un archivo simple.
2	**stats.isDirectory ()** Devuelve verdadero si el tipo de archivo de un directorio.
3	**stats.isBlockDevice ()** Devuelve verdadero si el tipo de archivo de un dispositivo de bloque.

4 4	**stats.isCharacterDevice ()** Devuelve verdadero si el tipo de archivo de un dispositivo de caracteres.
5 5	**stats.isSymbolicLink ()** Devuelve verdadero si el tipo de archivo de un enlace simbólico.
6 6	**stats.isFIFO ()** Devuelve verdadero si el tipo de archivo de un FIFO.
7 7	**stats.isSocket ()** Devuelve verdadero si el tipo de archivo de un asocket.

Ejemplo

Creemos un archivo js llamado **main.js** con el siguiente código:

```
var fs = require("fs");

console.log("Going to get file info!");
fs.stat('input.txt', function (err, stats) {
   if (err) {
      return console.error(err);
   }
   console.log(stats);
   console.log("Got file info successfully!");

   // Check file type
   console.log("isFile ? " + stats.isFile());
   console.log("isDirectory ? " + stats.isDirectory());
});
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Going to get file info!
{
   dev: 1792,
   mode: 33188,
   nlink: 1,
   uid: 48,
   gid: 48,
   rdev: 0,
   blksize: 4096,
   ino: 4318127,
   size: 97,
```

```
   blocks: 8,
   atime: Sun Mar 22 2015 13:40:00 GMT-0500 (CDT),
   mtime: Sun Mar 22 2015 13:40:57 GMT-0500 (CDT),
   ctime: Sun Mar 22 2015 13:40:57 GMT-0500 (CDT)
}
Got file info successfully!
isFile ? true
isDirectory ? false
```

Escribir un archivo

Sintaxis

A continuación se muestra la sintaxis de uno de los métodos para escribir en un archivo:

```
fs.writeFile(filename, data[, options], callback)
```

Este método sobrescribirá el archivo si el archivo ya existe. Si desea escribir en un archivo existente, entonces debe usar otro método disponible.

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **ruta** : esta es la cadena que tiene el nombre del archivo, incluida la ruta.
- **datos** : esta es la cadena o el búfer que se escribirá en el archivo.
- **opciones** : el tercer parámetro es un objeto que contendrá {codificación, modo, bandera}. Por defecto. la codificación es utf8, el modo es el valor octal 0666. y el indicador es 'w'
- **devolución de llamada** : esta es la función de devolución de llamada que obtiene un único error de parámetro que devuelve un error en caso de cualquier error de escritura.

Ejemplo

Vamos a crear un archivo js llamado **main.js que** tenga el siguiente código:

```
var fs = require("fs");

console.log("Going to write into existing file");
fs.writeFile('input.txt', 'Simply Easy Learning!',
function(err) {
   if (err) {
      return console.error(err);
   }

   console.log("Data written successfully!");
   console.log("Let's read newly written data");

   fs.readFile('input.txt', function (err, data) {
```

```
      if (err) {
         return console.error(err);
      }
      console.log("Asynchronous read: " + data.toString());
   });
});
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Going to write into existing file
Data written successfully!
Let's read newly written data
Asynchronous read: Simply Easy Learning!
```

Leer un archivo

Sintaxis

A continuación se muestra la sintaxis de uno de los métodos para leer un archivo:

```
fs.read(fd, buffer, offset, length, position, callback)
```

Este método usará el descriptor de archivo para leer el archivo. Si desea leer el archivo directamente con el nombre del archivo, debe usar otro método disponible.

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **fd** : este es el descriptor de archivo devuelto por fs.open ().
- **búfer** : este es el búfer en el que se escribirán los datos.
- **offset** : este es el desplazamiento en el búfer para comenzar a escribir.
- **longitud** : este es un número entero que especifica el número de bytes a leer.
- **position** : este es un número entero que especifica dónde comenzar a leer en el archivo. Si la posición es nula, los datos se leerán desde la posición actual del archivo.
- **devolución de llamada** : esta es la función de devolución de llamada que obtiene los tres argumentos (err, bytesRead, buffer).

Ejemplo

Creemos un archivo js llamado **main.js** con el siguiente código:

```
var fs = require("fs");
var buf = new Buffer(1024);
```

```
console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
      return console.error(err);
   }
   console.log("File opened successfully!");
   console.log("Going to read the file");

   fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
      if (err){
         console.log(err);
      }
      console.log(bytes + " bytes read");

      // Print only read bytes to avoid junk.
      if(bytes > 0){
         console.log(buf.slice(0, bytes).toString());
      }
   });
});
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Going to open an existing file
File opened successfully!
Going to read the file
97 bytes read
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!
```

Cerrar un archivo

Sintaxis

La siguiente es la sintaxis para cerrar un archivo abierto:

```
fs.close(fd, callback)
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **fd** : este es el descriptor de archivo devuelto por el método de archivo fs.open ().
- **devolución de llamada** : esta es la función de devolución de llamada. No existen argumentos distintos de una posible excepción para la devolución de llamada de finalización.

Ejemplo

Vamos a crear un archivo js llamado **main.js que** tenga el siguiente código:

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
      return console.error(err);
   }
   console.log("File opened successfully!");
   console.log("Going to read the file");

   fs.read(fd, buf, 0, buf.length, 0, function(err, bytes) {
      if (err) {
         console.log(err);
      }

      // Print only read bytes to avoid junk.
      if(bytes > 0) {
         console.log(buf.slice(0, bytes).toString());
      }

      // Close the opened file.
      fs.close(fd, function(err) {
         if (err) {
            console.log(err);
         }
         console.log("File closed successfully.");
      });
   });
});
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Going to open an existing file
File opened successfully!
Going to read the file
Postparaprogramadores is giving self learning content
to teach the world in simple and easy way!!!!!

File closed successfully.
```

Truncar un archivo

Sintaxis

A continuación se muestra la sintaxis del método para truncar un archivo abierto:

```
fs.ftruncate(fd, len, callback)
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **fd** : este es el descriptor de archivo devuelto por fs.open ().
- **len** : esta es la longitud del archivo después del cual el archivo se truncará.
- **devolución de llamada** : esta es la función de devolución de llamada. No existen argumentos distintos de una posible excepción para la devolución de llamada de finalización.

Ejemplo

Vamos a crear un archivo js llamado **main.js que** tenga el siguiente código:

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
      return console.error(err);
   }
   console.log("File opened successfully!");
   console.log("Going to truncate the file after 10 bytes");

   // Truncate the opened file.
   fs.ftruncate(fd, 10, function(err) {
      if (err) {
         console.log(err);
      }
      console.log("File truncated successfully.");
      console.log("Going to read the same file");

      fs.read(fd, buf, 0, buf.length, 0, function(err,
bytes){
         if (err) {
            console.log(err);
         }

         // Print only read bytes to avoid junk.
         if(bytes > 0) {
            console.log(buf.slice(0, bytes).toString());
         }

         // Close the opened file.
         fs.close(fd, function(err) {
            if (err) {
               console.log(err);
            }
            console.log("File closed successfully.");
```

```
         });
      });
   });
});
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Going to open an existing file
File opened successfully!
Going to truncate the file after 10 bytes
File truncated successfully.
Going to read the same file
Tutorials
File closed successfully.
```

Eliminar un archivo

Sintaxis

A continuación se muestra la sintaxis del método para eliminar un archivo:

```
fs.unlink(path, callback)
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **ruta** : este es el nombre del archivo, incluida la ruta.
- **devolución de llamada** : esta es la función de devolución de llamada. No existen argumentos distintos de una posible excepción para la devolución de llamada de finalización.

Ejemplo

Vamos a crear un archivo js llamado **main.js que** tenga el siguiente código:

```
var fs = require("fs");

console.log("Going to delete an existing file");
fs.unlink('input.txt', function(err) {
   if (err) {
      return console.error(err);
   }
   console.log("File deleted successfully!");
});
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Going to delete an existing file
File deleted successfully!
```

Crear un directorio

Sintaxis

A continuación se muestra la sintaxis del método para crear un directorio:

```
fs.mkdir(path[, mode], callback)
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **ruta** : este es el nombre del directorio, incluida la ruta.
- **modo** : este es el permiso de directorio que se debe configurar. El valor predeterminado es 0777.
- **devolución de llamada** : esta es la función de devolución de llamada. No existen argumentos distintos de una posible excepción para la devolución de llamada de finalización.

Ejemplo

Vamos a crear un archivo js llamado **main.js que** tenga el siguiente código:

```
var fs = require("fs");

console.log("Going to create directory /tmp/test");
fs.mkdir('/tmp/test',function(err) {
   if (err) {
      return console.error(err);
   }
   console.log("Directory created successfully!");
});
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Going to create directory /tmp/test
Directory created successfully!
```

Leer un directorio

Sintaxis

La siguiente es la sintaxis del método para leer un directorio:

```
fs.readdir(path, callback)
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **ruta** : este es el nombre del directorio, incluida la ruta.
- **devolución de llamada** : esta es la función de devolución de llamada que obtiene dos argumentos (err, archivos) donde archivos es una matriz de los nombres de los archivos en el directorio excluyendo '.' y '..'.

Ejemplo

Vamos a crear un archivo js llamado **main.js que** tenga el siguiente código:

```
var fs = require("fs");

console.log("Going to read directory /tmp");
fs.readdir("/tmp/",function(err, files) {
   if (err) {
      return console.error(err);
   }
   files.forEach( function (file) {
      console.log( file );
   });
});
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Going to read directory /tmp
ccmzx99o.out
ccyCSbkF.out
employee.ser
hsperfdata_apache
test
test.txt
```

Eliminar un directorio

Sintaxis

A continuación se muestra la sintaxis del método para eliminar un directorio:

```
fs.rmdir(path, callback)
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **ruta** : este es el nombre del directorio, incluida la ruta.

- **devolución de llamada** : esta es la función de devolución de llamada. No existen argumentos distintos de una posible excepción para la devolución de llamada de finalización.

Ejemplo

Vamos a crear un archivo js llamado **main.js que** tenga el siguiente código:

```
var fs = require("fs");

console.log("Going to delete directory /tmp/test");
fs.rmdir("/tmp/test",function(err) {
   if (err) {
      return console.error(err);
   }
   console.log("Going to read directory /tmp");

   fs.readdir("/tmp/",function(err, files) {
      if (err) {
         return console.error(err);
      }
      files.forEach( function (file) {
         console.log( file );
      });
   });
});
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida.

```
Going to read directory /tmp
ccmzx99o.out
ccyCSbkF.out
employee.ser
hsperfdata_apache
test.txt
```

Referencia de métodos

A continuación se incluye una referencia del módulo del sistema de archivos disponible en Node.js. Para más detalles puede consultar la documentación oficial.

Node.js - Objetos globales

Los objetos globales de Node.js son de naturaleza global y están disponibles en todos los módulos. No necesitamos incluir estos objetos en nuestra aplicación, sino que podemos usarlos directamente. Estos objetos son módulos, funciones, cadenas y el propio objeto como se explica a continuación.

__nombre del archivo

El __ nombre de archivo representa el nombre del archivo del código que se está ejecutando. Esta es la ruta absoluta resuelta de este archivo de código. Para un programa principal, este no es necesariamente el mismo nombre de archivo utilizado en la línea de comando. El valor dentro de un módulo es la ruta a ese archivo de módulo.

Ejemplo

Cree un archivo js llamado main.js con el siguiente código:

```
// Let's try to print the value of __filename

console.log( __filename );
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Según la ubicación de su programa, imprimirá el nombre del archivo principal de la siguiente manera:

```
/web/com/1427091028_21099/main.js
```

__dirname

El **__dirname** representa el nombre del directorio en el que reside el script que se está ejecutando actualmente.

Ejemplo

Cree un archivo js llamado main.js con el siguiente código:

```
// Let's try to print the value of __dirname

console.log( __dirname );
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Según la ubicación de su programa, imprimirá el nombre del directorio actual de la siguiente manera:

```
/web/com/1427091028_21099
```

setTimeout (cb, ms)

La función global **setTimeout (cb, ms)** se utiliza para ejecutar la devolución de llamada cb después de al menos ms milisegundos. El retraso real depende de factores externos como la granularidad del temporizador del sistema operativo y la carga del sistema. Un temporizador no puede abarcar más de 24.8 días.

Esta función devuelve un valor opaco que representa el temporizador que se puede utilizar para borrar el temporizador.

Ejemplo

Cree un archivo js llamado main.js con el siguiente código:

```
function printHello() {
   console.log( "Hello, World!");
}

// Now call above function after 2 seconds
setTimeout(printHello, 2000);
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique que la salida se imprima después de un pequeño retraso.

```
Hello, World!
```

clearTimeout (t)

La función global **clearTimeout (t)** se usa para detener un temporizador que se creó previamente con setTimeout (). Aquí **t** es el temporizador devuelto por la función setTimeout ().

Ejemplo

Cree un archivo js llamado main.js con el siguiente código:

```
function printHello() {
   console.log( "Hello, World!");
}

// Now call above function after 2 seconds
var t = setTimeout(printHello, 2000);

// Now clear the timer
clearTimeout(t);
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

Verifique la salida donde no encontrará nada impreso.

setInterval (cb, ms)

La función global **setInterval (cb, ms)** se usa para ejecutar la devolución de llamada cb repetidamente después de al menos ms milisegundos. El retraso

real depende de factores externos como la granularidad del temporizador del sistema operativo y la carga del sistema. Un temporizador no puede abarcar más de 24.8 días.

Esta función devuelve un valor opaco que representa el temporizador que se puede utilizar para borrar el temporizador utilizando la función **clearInterval (t)** .

Ejemplo

Cree un archivo js llamado main.js con el siguiente código:

```
function printHello() {
   console.log( "Hello, World!");
}

// Now call above function after 2 seconds
setInterval(printHello, 2000);
```

Ahora ejecute main.js para ver el resultado:

```
$ node main.js
```

El programa anterior ejecutará printHello () después de cada 2 segundos. Debido a limitaciones del sistema.

Objetos globales

La siguiente tabla proporciona una lista de otros objetos que usamos con frecuencia en nuestras aplicaciones. Para más detalles, puede consultar la documentación oficial.

No Señor.	Nombre y descripción del módulo
1	**Consola** Se utiliza para imprimir información en stdout y stderr.
2	**Proceso** Se utiliza para obtener información sobre el proceso actual. Proporciona múltiples eventos relacionados con actividades de proceso.

Node.js - Módulos de utilidad

Hay varios módulos de utilidad disponibles en la biblioteca de módulos Node.js. Estos módulos son muy comunes y se usan con frecuencia al desarrollar cualquier aplicación basada en Node.

No Señor.	Nombre y descripción del módulo
1	Módulo OS Proporciona funciones de utilidad relacionadas con el sistema operativo básico.
2	Módulo de ruta Proporciona utilidades para manejar y transformar rutas de archivos.
3	Módulo de red Proporciona servidores y clientes como flujos. Actúa como un contenedor de red.
4 4	Módulo DNS Proporciona funciones para realizar búsquedas DNS reales, así como para utilizar funcionalidades subyacentes de resolución de nombres del sistema operativo.
5 5	Módulo de dominio Proporciona formas de manejar múltiples operaciones de E / S diferentes como un solo grupo.

Node.js - Módulo web

¿Qué es un servidor Web?

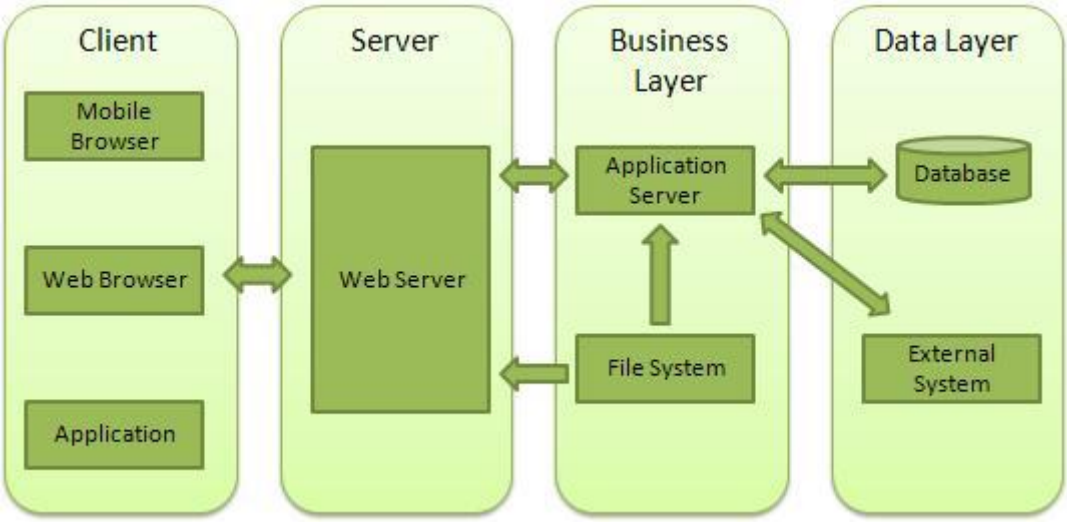
Un servidor web es una aplicación de software que maneja las solicitudes HTTP enviadas por el cliente HTTP, como los navegadores web, y devuelve páginas web en respuesta a los clientes. Los servidores web generalmente entregan documentos html junto con imágenes, hojas de estilo y scripts.

La mayoría de los servidores web admiten scripts del lado del servidor, utilizan lenguajes de scripting o redirigen la tarea a un servidor de aplicaciones que recupera datos de una base de datos y realiza una lógica compleja y luego envía un resultado al cliente HTTP a través del servidor web.

El servidor web Apache es uno de los servidores web más utilizados. Es un proyecto de código abierto.

Arquitectura de aplicaciones web

Una aplicación web generalmente se divide en cuatro capas:



- **Cliente** : esta capa consta de navegadores web, navegadores móviles o aplicaciones que pueden realizar solicitudes HTTP al servidor web.
- **Servidor** : esta capa tiene el servidor web que puede interceptar las solicitudes realizadas por los clientes y transmitirles la respuesta.
- **Empresa** : esta capa contiene el servidor de aplicaciones que utiliza el servidor web para realizar el procesamiento requerido. Esta capa interactúa con la capa de datos a través de la base de datos o algunos programas externos.
- **Datos** : esta capa contiene las bases de datos o cualquier otra fuente de datos.

Crear un servidor web usando el nodo

Node.js proporciona un módulo **http** que se puede usar para crear un cliente HTTP de un servidor. A continuación se muestra la estructura mínima del servidor HTTP que escucha en el puerto 8081.

Cree un archivo js llamado server.js -

Archivo: server.js

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

// Create a server
http.createServer( function (request, response) {
   // Parse the request containing file name
   var pathname = url.parse(request.url).pathname;

   // Print the name of the file for which request is made.
   console.log("Request for " + pathname + " received.");

   // Read the requested file content from file system
   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {
         console.log(err);

         // HTTP Status: 404 : NOT FOUND
```

```
         // Content Type: text/plain
         response.writeHead(404, {'Content-Type':
'text/html'});
      } else {
         //Page found
         // HTTP Status: 200 : OK
         // Content Type: text/plain
         response.writeHead(200, {'Content-Type':
'text/html'});

         // Write the content of the file to response body
         response.write(data.toString());
      }

      // Send the response body
      response.end();
   });
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

A continuación, creemos el siguiente archivo html llamado index.htm en el mismo directorio donde creó server.js.

Archivo: index.htm

```
<html>
   <head>
      <title>Sample Page</title>
   </head>

   <body>
      Hello World!
   </body>
</html>
```

Ahora ejecutemos el server.js para ver el resultado:

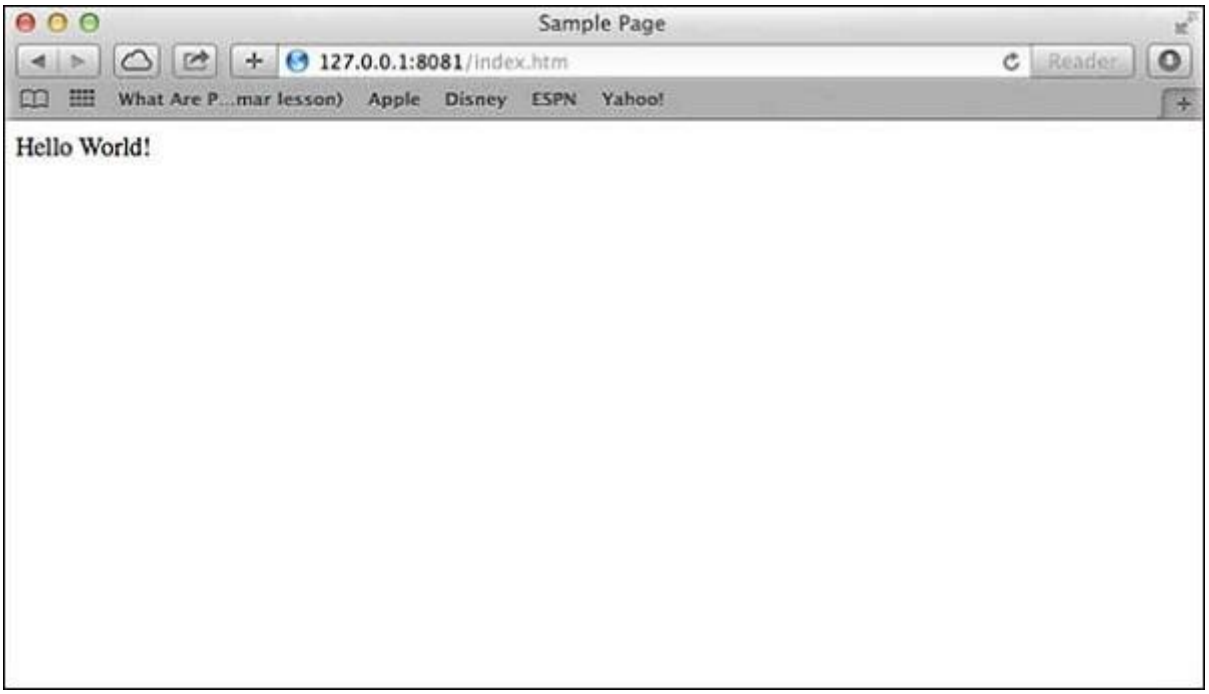
```
$ node server.js
```

Verifique la salida.

```
Server running at http://127.0.0.1:8081/
```

Hacer una solicitud al servidor Node.js

Abra http://127.0.0.1:8081/index.htm en cualquier navegador para ver el siguiente resultado.



Verifique la salida al final del servidor.

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.
```

Crear cliente web usando Nodo

Se puede crear un cliente web utilizando el módulo **http** . Veamos el siguiente ejemplo.

Cree un archivo js llamado client.js -

Archivo: client.js

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

// Options to be used by request
var options = {
   host: 'localhost',
   port: '8081',
   path: '/index.htm'
};

// Callback function is used to deal with response
var callback = function(response) {
   // Continuously update stream with data
   var body = '';
   response.on('data', function(data) {
      body += data;
   });

   response.on('end', function() {
      // Data received completely.
      console.log(body);
```

```
   });
}
// Make a request to the server
var req = http.request(options, callback);
req.end();
```

Ahora ejecute el client.js desde un terminal de comando diferente que no sea server.js para ver el resultado:

```
$ node client.js
```

Verifique la salida.

```
<html>
   <head>
      <title>Sample Page</title>
   </head>

   <body>
      Hello World!
   </body>
</html>
```

Verifique la salida al final del servidor.

```
Server running at http://127.0.0.1:8081/
Request for /index.htm received.
```

Node.js - Express Framework

Resumen expreso

Express es un marco de aplicación web Node.js mínimo y flexible que proporciona un conjunto robusto de características para desarrollar aplicaciones web y móviles. Facilita el rápido desarrollo de aplicaciones web basadas en nodos. Las siguientes son algunas de las características principales de Express Framework:

- Permite configurar middlewares para responder a solicitudes HTTP.
- Define una tabla de enrutamiento que se utiliza para realizar diferentes acciones según el método HTTP y la URL.
- Permite renderizar dinámicamente páginas HTML basadas en pasar argumentos a plantillas.

Instalando Express

Primero, instale el framework Express globalmente usando NPM para que pueda usarse para crear una aplicación web usando un terminal de nodo.

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

El comando anterior guarda la instalación localmente en el directorio **node_modules** y crea un directorio express dentro de

node_modules. Debe instalar los siguientes módulos importantes junto con express:

- **body-parser** : este es un middleware de node.js para manejar datos de formulario codificados JSON, Raw, Text y URL.
- **cookie-parser** : analiza el encabezado de Cookie y rellena req.cookies con un objeto marcado por los nombres de las cookies.
- **multer** : este es un middleware de node.js para manejar datos multiparte / formulario.

```
$ npm install body-parser --save
$ npm install cookie-parser --save
$ npm install multer --save
```

Hola mundo ejemplo

A continuación, se incluye una aplicación Express muy básica que inicia un servidor y escucha la conexión en el puerto 8081. Esta aplicación responde con **Hello World!** para solicitudes a la página de inicio. Para cualquier otra ruta, responderá con un **404 No encontrado.**

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

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Example app listening at http://%s:%s", host,
port)
})
```

Guarde el código anterior en un archivo llamado server.js y ejecútelo con el siguiente comando.

```
$ node server.js
```

Verá el siguiente resultado:

```
Example app listening at http://0.0.0.0:8081
```

Abra http://127.0.0.1:8081/ en cualquier navegador para ver el siguiente resultado.

Solicitar respuesta

La aplicación Express utiliza una función de devolución de llamada cuyos parámetros son objetos de **solicitud** y **respuesta** .

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

- Objeto de solicitud: el objeto de solicitud representa la solicitud HTTP y tiene propiedades para la cadena de consulta de solicitud, los parámetros, el cuerpo, los encabezados HTTP, etc.
- Objeto de respuesta: el objeto de respuesta representa la respuesta HTTP que envía una aplicación Express cuando recibe una solicitud HTTP.

Puede imprimir objetos **req** y **res** que proporcionan mucha información relacionada con la solicitud y respuesta HTTP, incluidas cookies, sesiones, URL, etc.

Enrutamiento Básico

Hemos visto una aplicación básica que atiende solicitudes HTTP para la página de inicio. El enrutamiento se refiere a determinar cómo una aplicación responde a una solicitud del cliente a un punto final particular, que es un URI (o ruta) y un método de solicitud HTTP específico (GET, POST, etc.).

Extendiremos nuestro programa Hello World para manejar más tipos de solicitudes HTTP.

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

// This responds with "Hello World" on the homepage
```

```
app.get('/', function (req, res) {
   console.log("Got a GET request for the homepage");
   res.send('Hello GET');
})

// This responds a POST request for the homepage
app.post('/', function (req, res) {
   console.log("Got a POST request for the homepage");
   res.send('Hello POST');
})

// This responds a DELETE request for the /del_user page.
app.delete('/del_user', function (req, res) {
   console.log("Got a DELETE request for /del_user");
   res.send('Hello DELETE');
})

// This responds a GET request for the /list_user page.
app.get('/list_user', function (req, res) {
   console.log("Got a GET request for /list_user");
   res.send('Page Listing');
})

// This responds a GET request for abcd, abxcd, ab123cd, and
so on
app.get('/ab*cd', function(req, res) {
   console.log("Got a GET request for /ab*cd");
   res.send('Page Pattern Match');
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Example app listening at http://%s:%s", host,
port)
})
```

Guarde el código anterior en un archivo llamado server.js y ejecútelo con el siguiente comando.

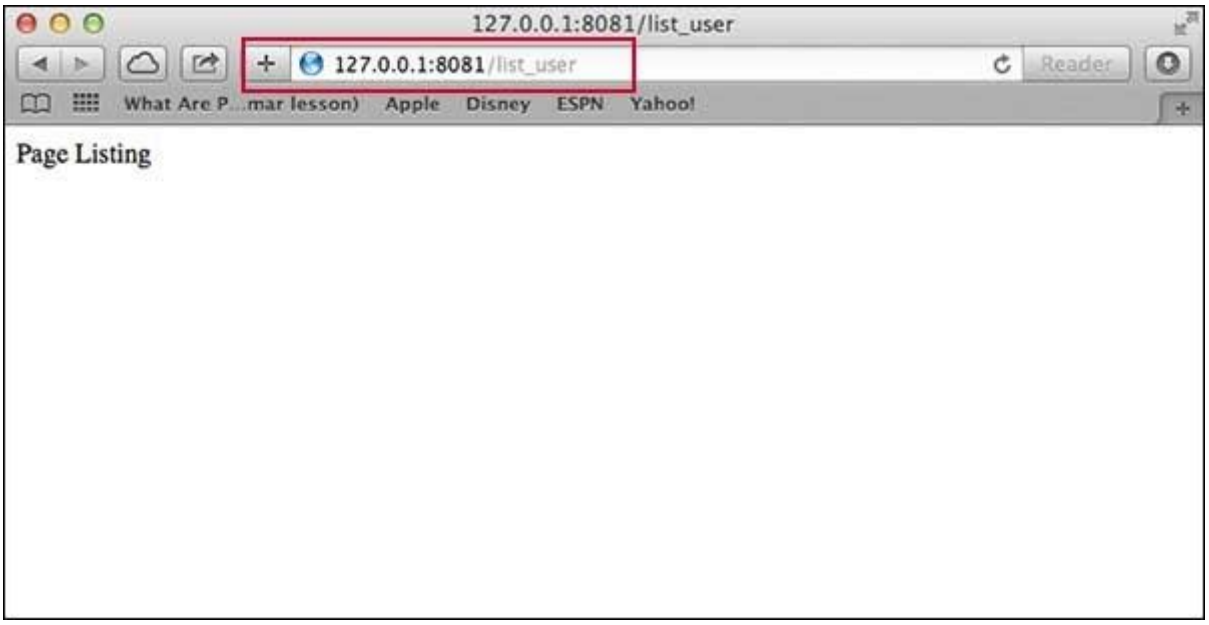
```
$ node server.js
```

Verá el siguiente resultado:

```
Example app listening at http://0.0.0.0:8081
```

Ahora puede probar diferentes solicitudes en http://127.0.0.1:8081 para ver la salida generada por server.js. A continuación se presentan algunas capturas de pantalla que muestran diferentes respuestas para diferentes URL.

Pantalla que muestra nuevamente http://127.0.0.1:8081/list_user



Pantalla que muestra nuevamente http://127.0.0.1:8081/abcd

Pantalla que muestra nuevamente http://127.0.0.1:8081/abcdefg

Sirviendo archivos estáticos

Express proporciona un middleware incorporado **express.static** para servir archivos estáticos, como imágenes, CSS, JavaScript, etc.

Simplemente necesita pasar el nombre del directorio donde guarda sus activos estáticos, al middleware **express.static** para comenzar a servir los archivos directamente. Por ejemplo, si mantiene sus archivos de imágenes, CSS y JavaScript en un directorio llamado público, puede hacer esto:

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

Mantendremos algunas imágenes en **el** subdirectorio **public / images** de la siguiente manera:

```
node_modules
server.js
public/
public/images
public/images/logo.png
```

Modifiquemos la aplicación "Hello Word" para agregar la funcionalidad para manejar archivos estáticos.

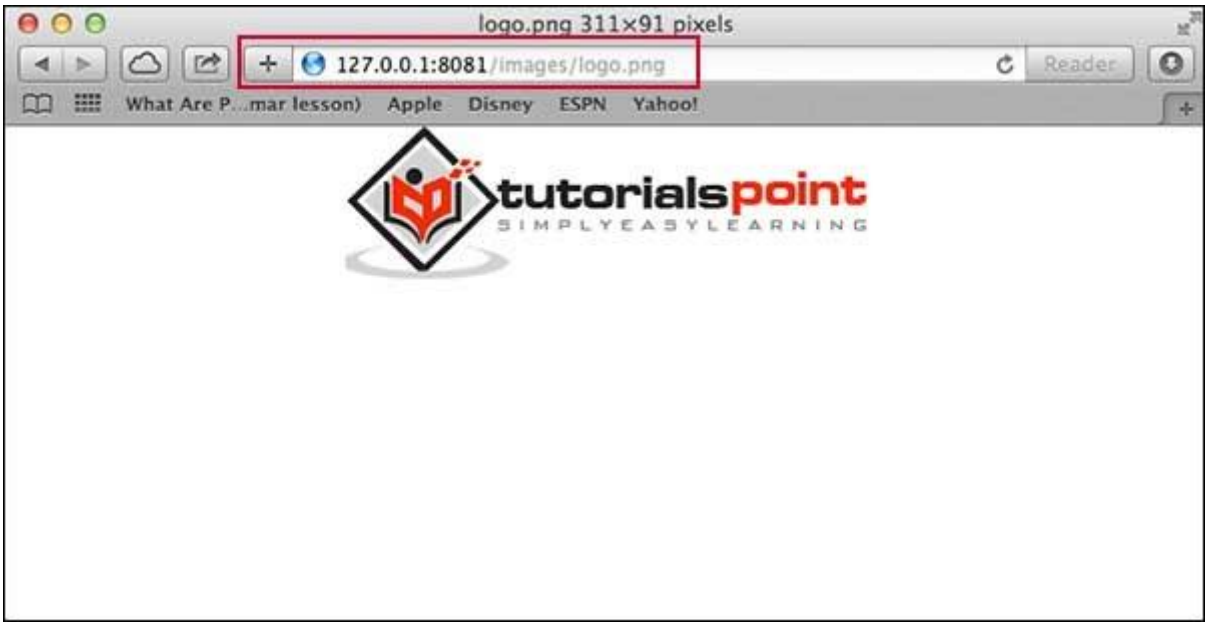
```
var express = require('express');
var app = express();

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

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port
```

```
   console.log("Example app listening at http://%s:%s", host,
port)
})
```

Guarde el código anterior en un archivo llamado server.js y ejecútelo con el siguiente comando.

```
$ node server.js
```

Ahora abra http://127.0.0.1:8081/images/logo.png en cualquier navegador y vea el siguiente resultado.



OBTENER Método

Aquí hay un ejemplo simple que pasa dos valores usando el método HTML FORM GET. Vamos a usar el router **process_get** dentro de server.js para manejar esta entrada.

```
<html>
   <body>

      <form action = "http://127.0.0.1:8081/process_get"
method = "GET">
         First Name: <input type = "text" name =
"first_name">  <br>
         Last Name: <input type = "text" name = "last_name">
         <input type = "submit" value = "Submit">
      </form>

   </body>
</html>
```

Guardemos el código anterior en index.htm y modifique server.js para manejar las solicitudes de la página de inicio, así como la entrada enviada por el formulario HTML.

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

app.use(express.static('public'));
app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.get('/process_get', function (req, res) {
   // Prepare output in JSON format
   response = {
      first_name:req.query.first_name,
      last_name:req.query.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Example app listening at http://%s:%s", host,
port)
})
```

Acceder al documento HTML usando *http://127.0.0.1:8081/index.htm* generará el siguiente formulario:

First Name:

Last Name:

Ahora puede ingresar el nombre y apellido y luego hacer clic en el botón Enviar para ver el resultado y debería devolver el siguiente resultado:

```
{"first_name":"John","last_name":"Paul"}
```

Método POST

Aquí hay un ejemplo simple que pasa dos valores usando el método HTML FORM POST. Vamos a usar el router **process_get** dentro de server.js para manejar esta entrada.

```
<html>
   <body>
```

```
      <form action = "http://127.0.0.1:8081/process_post"
method = "POST">
         First Name: <input type = "text" name =
"first_name"> <br>
         Last Name: <input type = "text" name = "last_name">
         <input type = "submit" value = "Submit">
      </form>

   </body>
</html>
```

Guardemos el código anterior en index.htm y modifique server.js para manejar las solicitudes de la página de inicio, así como la entrada enviada por el formulario HTML.

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

// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended:
false })

app.use(express.static('public'));
app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/process_post', urlencodedParser, function (req,
res) {
   // Prepare output in JSON format
   response = {
      first_name:req.body.first_name,
      last_name:req.body.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Example app listening at http://%s:%s", host,
port)
})
```

Acceder al documento HTML usando *http://127.0.0.1:8081/index.htm* generará el siguiente formulario:

First Name:

Last Name:

Ahora puede ingresar el nombre y apellido y luego hacer clic en el botón enviar para ver el siguiente resultado:

```
{"first_name":"John","last_name":"Paul"}
```

Subir archivo

El siguiente código HTML crea un formulario de carga de archivos. Este formulario tiene el atributo de método establecido en **POST** y el atributo enctype está configurado en **multipart / form-data**

```
<html>
   <head>
      <title>File Uploading Form</title>
   </head>

   <body>
      <h3>File Upload:</h3>
      Select a file to upload: <br />

      <form action = "http://127.0.0.1:8081/file_upload"
method = "POST"
         enctype = "multipart/form-data">
         <input type="file" name="file" size="50" />
         <br />
         <input type = "submit" value = "Upload File" />
      </form>

   </body>
</html>
```

Guardemos el código anterior en index.htm y modifique server.js para manejar las solicitudes de la página de inicio, así como la carga de archivos.

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

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

app.use(express.static('public'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(multer({ dest: '/tmp/'}));

app.get('/index.htm', function (req, res) {
   res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/file_upload', function (req, res) {
   console.log(req.files.file.name);
```

```
   console.log(req.files.file.path);
   console.log(req.files.file.type);
   var file = __dirname + "/" + req.files.file.name;

   fs.readFile( req.files.file.path, function (err, data) {
      fs.writeFile(file, data, function (err) {
         if( err ) {
            console.log( err );
            } else {
               response = {
                  message:'File uploaded successfully',
                  filename:req.files.file.name
               };
            }

         console.log( response );
         res.end( JSON.stringify( response ) );
      });
   });
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Example app listening at http://%s:%s", host,
port)
})
```

Acceder al documento HTML usando *http://127.0.0.1:8081/index.htm* generará el siguiente formulario:

File Upload:
Select a file to upload:

NOTE: This is just dummy form and would not work, but it must work at your server.

Manejo de cookies

Puede enviar cookies a un servidor Node.js que puede manejar lo mismo utilizando la siguiente opción de middleware. El siguiente es un ejemplo simple para imprimir todas las cookies enviadas por el cliente.

```
var express      = require('express')
var cookieParser = require('cookie-parser')

var app = express()
app.use(cookieParser())
```

```
app.get('/', function(req, res) {
   console.log("Cookies: ", req.cookies)
})
app.listen(8081)
```

Node.js - API RESTful

¿Qué es la arquitectura REST?

REST significa transferencia de estado representativa. REST es una arquitectura basada en estándares web y utiliza el protocolo HTTP. Gira en torno al recurso donde cada componente es un recurso y se accede a un recurso mediante una interfaz común utilizando métodos estándar HTTP. REST fue presentado por primera vez por Roy Fielding en 2000.

Un servidor REST simplemente proporciona acceso a los recursos y el cliente REST accede y modifica los recursos mediante el protocolo HTTP. Aquí cada recurso se identifica mediante URI / ID globales. REST usa varias representaciones para representar un recurso como texto, JSON, XML, pero JSON es el más popular.

Métodos HTTP

Los siguientes cuatro métodos HTTP se usan comúnmente en la arquitectura basada en REST.

- **GET** : se utiliza para proporcionar un acceso de solo lectura a un recurso.
- **PUT** : se usa para crear un nuevo recurso.
- **BORRAR** : se utiliza para eliminar un recurso.
- **POST** : se utiliza para actualizar un recurso existente o crear un nuevo recurso.

Servicios web RESTful

Un servicio web es una colección de protocolos y estándares abiertos que se utilizan para intercambiar datos entre aplicaciones o sistemas. Las aplicaciones de software escritas en varios lenguajes de programación y que se ejecutan en varias plataformas pueden usar servicios web para intercambiar datos a través de redes informáticas como Internet de manera similar a la comunicación entre procesos en una sola computadora. Esta interoperabilidad (p. Ej., Comunicación entre Java y Python, o aplicaciones de Windows y Linux) se debe al uso de estándares abiertos.

Los servicios web basados en la arquitectura REST se conocen como servicios web RESTful. Estos servicios web utilizan métodos HTTP para implementar el concepto de arquitectura REST. Un servicio web RESTful generalmente define un URI, Identificador uniforme de recursos, un servicio, que proporciona representación de recursos como JSON y un conjunto de métodos HTTP.

Crear RESTful para una biblioteca

Considere que tenemos una base de datos basada en JSON de usuarios que tiene los siguientes usuarios en un archivo **users.json** :

```
{
   "user1" : {
      "name" : "mahesh",
      "password" : "password1",
      "profession" : "teacher",
      "id": 1
   },

   "user2" : {
      "name" : "suresh",
      "password" : "password2",
      "profession" : "librarian",
      "id": 2
   },

   "user3" : {
      "name" : "ramesh",
      "password" : "password3",
      "profession" : "clerk",
      "id": 3
   }
}
```

En base a esta información, proporcionaremos las siguientes API RESTful.

No Señor.	URI	Método HTTP	POST cuerpo	Resultado
1	listUsers	OBTENER	vacío	Mostrar lista de todos los usuarios.
2	agregar usuario	ENVIAR	JSON String	Agregar detalles del nuevo usuario.
3	borrar usuario	ELIMINAR	JSON String	Eliminar un usuario existente.
4 4	:carné de identidad	OBTENER	vacío	Mostrar detalles de un usuario.

Mantengo la mayor parte de todos los ejemplos en forma de codificación rígida, suponiendo que ya sepa cómo pasar valores desde el front-end usando Ajax o datos de formularios simples y cómo procesarlos usando el objeto **Solicitud** expresa .

Lista de usuarios

Implementemos nuestro primer RESTful API **listUsers** usando el siguiente código en un archivo server.js:

server.js

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

app.get('/listUsers', function (req, res) {
   fs.readFile( __dirname + "/" + "users.json", 'utf8',
function (err, data) {
      console.log( data );
      res.end( data );
   });
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port
   console.log("Example app listening at http://%s:%s", host,
port)
})
```

Ahora intente acceder a la API definida usando la *URL: http://127.0.0.1:8081/listUsers* y *Método HTTP: OBTENGA* en la máquina local usando cualquier cliente REST. Esto debería producir el siguiente resultado:

Puede cambiar la dirección IP dada cuando colocará la solución en el entorno de producción.

```
{
   "user1" : {
      "name" : "mahesh",
      "password" : "password1",
      "profession" : "teacher",
      "id": 1
   },

   "user2" : {
      "name" : "suresh",
      "password" : "password2",
      "profession" : "librarian",
      "id": 2
   },

   "user3" : {
      "name" : "ramesh",
      "password" : "password3",
      "profession" : "clerk",
      "id": 3
   }
```

```
}
```

Agregar usuario

La siguiente API le mostrará cómo agregar un nuevo usuario a la lista. A continuación se detallan los nuevos usuarios:

```
user = {
   "user4" : {
      "name" : "mohit",
      "password" : "password4",
      "profession" : "teacher",
      "id": 4
   }
}
```

Puede aceptar la misma entrada en forma de JSON utilizando la llamada Ajax, pero para enseñar el punto de vista, aquí lo estamos codificando. A continuación se **muestra la** API **addUser** a un nuevo usuario en la base de datos:

server.js

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

var user = {
   "user4" : {
      "name" : "mohit",
      "password" : "password4",
      "profession" : "teacher",
      "id": 4
   }
}

app.post('/addUser', function (req, res) {
   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8',
function (err, data) {
      data = JSON.parse( data );
      data["user4"] = user["user4"];
      console.log( data );
      res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port
   console.log("Example app listening at http://%s:%s", host,
port)
})
```

Ahora intente acceder a la API definida usando la *URL: http://127.0.0.1:8081/addUser* y *Método HTTP: POST* en la máquina local usando cualquier cliente REST. Esto debería producir el siguiente resultado:

```
{

"user1":{"name":"mahesh","password":"password1","profession":
"teacher","id":1},

"user2":{"name":"suresh","password":"password2","profession":
"librarian","id":2},

"user3":{"name":"ramesh","password":"password3","profession":
"clerk","id":3},

"user4":{"name":"mohit","password":"password4","profession":"
teacher","id":4}
}
```

Mostrar detalle

Ahora implementaremos una API que se llamará utilizando la ID de usuario y mostrará los detalles del usuario correspondiente.

server.js

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

app.get('/:id', function (req, res) {
   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8',
function (err, data) {
      var users = JSON.parse( data );
      var user = users["user" + req.params.id]
      console.log( user );
      res.end( JSON.stringify(user));
   });
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port
   console.log("Example app listening at http://%s:%s", host,
port)
})
```

Ahora intente acceder a la API definida usando la *URL: http://127.0.0.1:8081/2* y *Método HTTP: OBTENGA* en la máquina local usando cualquier cliente REST. Esto debería producir el siguiente resultado:

```
{"name":"suresh","password":"password2","profession":"librari
an","id":2}
```

Borrar usuario

Esta API es muy similar a la API addUser, donde recibimos datos de entrada a través de req.body y luego, según la ID de usuario, eliminamos a ese usuario de la base de datos. Para mantener nuestro programa simple, asumimos que vamos a eliminar al usuario con ID 2.

server.js

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

var id = 2;

app.delete('/deleteUser', function (req, res) {
   // First read existing users.
   fs.readFile( __dirname + "/" + "users.json", 'utf8',
function (err, data) {
      data = JSON.parse( data );
      delete data["user" + 2];

      console.log( data );
      res.end( JSON.stringify(data));
   });
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port
   console.log("Example app listening at http://%s:%s", host,
port)
})
```

Ahora intente acceder a la API definida usando la *URL: http://127.0.0.1:8081/deleteUser* y *Método HTTP: DELETE* en la máquina local usando cualquier cliente REST. Esto debería producir el siguiente resultado:

```
{"user1":{"name":"mahesh","password":"password1","profession"
:"teacher","id":1},
"user3":{"name":"ramesh","password":"password3","profession":
"clerk","id":3}}
```

Node.js - Aplicación de escalado

Node.js se ejecuta en un modo de subproceso único, pero utiliza un paradigma controlado por eventos para manejar la concurrencia. También facilita la creación de procesos secundarios para aprovechar el procesamiento paralelo en sistemas basados en CPU multinúcleo.

Los procesos **secundarios** siempre tienen tres secuencias **child.stdin** , **child.stdout** y **child.stderr** que pueden compartirse con las secuencias stdio del proceso padre.

Node proporciona **un** módulo **child_process** que tiene las siguientes tres formas principales de crear un proceso hijo.

- **exec** : el método child_process.exec ejecuta un comando en un shell / consola y almacena el resultado en la memoria intermedia.
- **spawn** - child_process.spawn inicia un nuevo proceso con un comando dado.
- **fork** : el método child_process.fork es un caso especial de spawn () para crear procesos secundarios.

El método exec ()

El método child_process.exec ejecuta un comando en un shell y almacena el resultado. Tiene la siguiente firma:

```
child_process.exec(command[, options], callback)
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **command** (String) El comando a ejecutar, con argumentos separados por espacios
- **Las opciones** (Objeto) pueden comprender una o más de las siguientes opciones:
  - **cwd** (String) Directorio de trabajo actual del proceso hijo
  - **Env** (Objeto) Entorno clave-valor pares
  - **codificación** (Cadena) (Valor predeterminado: 'utf8')
  - **shell** (String) Shell para ejecutar el comando con (Predeterminado: '/ bin / sh' en UNIX, 'cmd.exe' en Windows, El shell debe comprender el modificador -c en UNIX o / s / c en Windows. En Windows , el análisis de la línea de comandos debe ser compatible con cmd.exe).
  - **tiempo de espera** (Número) (Predeterminado: 0)
  - **maxBuffer** (Number) (Predeterminado: 200 * 1024)
  - **killSignal** (String) (Predeterminado: 'SIGTERM')
  - **uid** (Number) Establece la identidad del usuario del proceso.
  - **gid** (Number) Establece la identidad del grupo del proceso.
- **devolución de llamada** La función obtiene tres argumentos **error** , **stdout** y **stderr** que se llaman con la salida cuando finaliza el proceso.

El método exec () devuelve un búfer con un tamaño máximo y espera a que finalice el proceso e intenta devolver todos los datos almacenados a la vez.

Ejemplo

Vamos a crear dos archivos js llamados support.js y master.js -

Archivo: support.js

```
console.log("Child Process " + process.argv[2] + " executed."
);
```

Archivo: master.js

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

for(var i=0; i<3; i++) {
   var workerProcess = child_process.exec('node support.js
'+i,function
      (error, stdout, stderr) {

      if (error) {
         console.log(error.stack);
         console.log('Error code: '+error.code);
         console.log('Signal received: '+error.signal);
      }
      console.log('stdout: ' + stdout);
      console.log('stderr: ' + stderr);
   });

   workerProcess.on('exit', function (code) {
      console.log('Child process exited with exit code
'+code);
   });
}
```

Ahora ejecute master.js para ver el resultado:

```
$ node master.js
```

Verifique la salida. El servidor ha comenzado.

```
Child process exited with exit code 0
stdout: Child Process 1 executed.

stderr:
Child process exited with exit code 0
stdout: Child Process 0 executed.

stderr:
Child process exited with exit code 0
stdout: Child Process 2 executed.
```

El método spawn ()

El método child_process.spawn inicia un nuevo proceso con un comando dado. Tiene la siguiente firma:

```
child_process.spawn(command[, args][, options])
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **command** (String) El comando para ejecutar
- **args** (Array) Lista de argumentos de cadena
- **Las opciones** (Objeto) pueden comprender una o más de las siguientes opciones:
    - **cwd** (String) Directorio de trabajo actual del proceso hijo.
    - **env** (Objeto) Entorno clave-valor pares.
    - **stdio** (Array) String Configuración stdio del niño.
    - **Descriptores de** archivos obsoletos **customFds** (matriz) para que el niño los use para stdio.
    - **separado** (booleano) El niño será un líder de grupo de proceso.
    - **uid** (Number) Establece la identidad del usuario del proceso.
    - **gid** (Number) Establece la identidad del grupo del proceso.

El método spawn () devuelve streams (stdout & stderr) y debe usarse cuando el proceso devuelve una cantidad de datos en volumen. spawn () comienza a recibir la respuesta tan pronto como el proceso comienza a ejecutarse.

Ejemplo

Cree dos archivos js llamados support.js y master.js:

Archivo: support.js

```
console.log("Child Process " + process.argv[2] + " executed."
);
```

Archivo: master.js

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

for(var i = 0; i<3; i++) {
   var workerProcess = child_process.spawn('node',
['support.js', i]);

   workerProcess.stdout.on('data', function (data) {
      console.log('stdout: ' + data);
   });

   workerProcess.stderr.on('data', function (data) {
      console.log('stderr: ' + data);
   });

   workerProcess.on('close', function (code) {
      console.log('child process exited with code ' + code);
   });
}
```

Ahora ejecute master.js para ver el resultado:

```
$ node master.js
```

Verifique la salida. El servidor ha comenzado

```
stdout: Child Process 0 executed.

child process exited with code 0
stdout: Child Process 1 executed.

stdout: Child Process 2 executed.

child process exited with code 0
child process exited with code 0
```

El método fork ()

El método child_process.fork es un caso especial de spawn () para crear procesos Node. Tiene la siguiente firma:

```
child_process.fork(modulePath[, args][, options])
```

Parámetros

Aquí está la descripción de los parámetros utilizados:

- **modulePath** (String) El módulo a ejecutar en el hijo.
- **args** (Array) Lista de argumentos de cadena
- **Las opciones** (Objeto) pueden comprender una o más de las siguientes opciones:
    - **cwd** (String) Directorio de trabajo actual del proceso hijo.
    - **env** (Objeto) Entorno clave-valor pares.
    - **execPath** (String) Ejecutable utilizado para crear el proceso hijo.
    - **execArgv** (Array) Lista de argumentos de cadena pasados al ejecutable (Predeterminado: process.execArgv).
    - **silent** (booleano) Si es verdadero, stdin, stdout y stderr del elemento secundario se canalizarán al elemento primario; de lo contrario, se heredarán del elemento primario, consulte las opciones "pipe" y "heredar" para el elemento de spawn () para más detalles (el valor predeterminado es falso).
    - **uid** (Number) Establece la identidad del usuario del proceso.
    - **gid** (Number) Establece la identidad del grupo del proceso.

El método fork devuelve un objeto con un canal de comunicación incorporado además de tener todos los métodos en una instancia ChildProcess normal.

Ejemplo

Cree dos archivos js llamados support.js y master.js:

Archivo: support.js

```
console.log("Child Process " + process.argv[2] + " executed."
);
```

Archivo: master.js

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

for(var i=0; i<3; i++) {
   var worker_process = child_process.fork("support.js",
[i]);

   worker_process.on('close', function (code) {
      console.log('child process exited with code ' + code);
   });
}
```

Ahora ejecute master.js para ver el resultado:

```
$ node master.js
```

Verifique la salida. El servidor ha comenzado.

```
Child Process 0 executed.
Child Process 1 executed.
Child Process 2 executed.
child process exited with code 0
child process exited with code 0
child process exited with code 0
```

Node.js - Embalaje

JXcore, que es un proyecto de código abierto, presenta una característica única para empaquetar y cifrar archivos fuente y otros activos en paquetes JX.

Considere que tiene un gran proyecto que consta de muchos archivos. JXcore puede empaquetarlos a todos en un solo archivo para simplificar la distribución. Este capítulo proporciona una descripción general rápida de todo el proceso a partir de la instalación de JXcore.

Instalación JXcore

Instalar JXcore es bastante simple. Aquí hemos proporcionado instrucciones paso a paso sobre cómo instalar JXcore en su sistema. Siga los pasos que se detallan a continuación:

Paso 1

Descargue el paquete JXcore de https://github.com/jxcore/jxcore , según su sistema operativo y la arquitectura de la máquina. Descargamos un paquete para Cenots que se ejecuta en una máquina de 64 bits.

```
$ wget https://s3.amazonaws.com/nodejx/jx_rh64.zip
```

Paso 2

Descomprima el archivo descargado **jx_rh64.zip** y copie el binario **jx** en / usr / bin o puede estar en cualquier otro directorio basado en la configuración de su sistema.

```
$ unzip jx_rh64.zip
$ cp jx_rh64/jx /usr/bin
```

Paso 3

Establezca su variable PATH adecuadamente para ejecutar jx desde cualquier lugar que desee.

```
$ export PATH=$PATH:/usr/bin
```

Etapa 4

Puede verificar su instalación emitiendo un comando simple como se muestra a continuación. Debería encontrarlo funcionando e imprimiendo su número de versión de la siguiente manera:

```
$ jx --version
v0.10.32
```

Empaquetando el Código

Considere que tiene un proyecto con los siguientes directorios donde guardó todos sus archivos, incluidos Node.js, archivo principal, index.js y todos los módulos instalados localmente.

```
drwxr-xr-x  2 root root  4096 Nov 13 12:42 images
-rwxr-xr-x  1 root root 30457 Mar  6 12:19 index.htm
-rwxr-xr-x  1 root root 30452 Mar  1 12:54 index.js
drwxr-xr-x 23 root root  4096 Jan 15 03:48 node_modules
drwxr-xr-x  2 root root  4096 Mar 21 06:10 scripts
drwxr-xr-x  2 root root  4096 Feb 15 11:56 style
```

Para empaquetar el proyecto anterior, simplemente necesita ir dentro de este directorio y emitir el siguiente comando jx. Suponiendo que index.js es el archivo de entrada para su proyecto Node.js:

```
$ jx package index.js index
```

Aquí podría haber usado cualquier otro nombre de paquete en lugar de **índice.** Hemos utilizado **index** porque queríamos mantener nuestro nombre de archivo principal como index.jx. Sin embargo, el comando anterior empacará todo y creará los siguientes dos archivos:

- **index.jxp** Este es un archivo intermedio que contiene los detalles completos del proyecto necesarios para compilar el proyecto.
- **index.jx** Este es el archivo binario que tiene el paquete completo que está listo para ser enviado a su cliente o su entorno de producción.

Lanzamiento de archivo JX

Considere que su proyecto original de Node.js se ejecutaba de la siguiente manera:

```
$ node index.js command_line_arguments
```

Después de compilar su paquete usando JXcore, se puede iniciar de la siguiente manera:

```
$ jx index.jx command_line_arguments
```

Para saber más sobre JXcore, puede consultar su sitio web oficial.