EBook Gratuito

APPRENDIMENTO Node.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#node.js

Sommario

Di

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

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

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

Capitolo 1: Iniziare con Node.js

Osservazioni

Node.js è un framework I / O asincrono basato su eventi, non bloccante, che utilizza il motore JavaScript V8 di Google. Viene utilizzato per lo sviluppo di applicazioni che sfruttano pesantemente la possibilità di eseguire JavaScript sia sul client, sia sul lato server e quindi beneficiano della riutilizzabilità del codice e della mancanza di cambio di contesto. È open source e multipiattaforma. Le applicazioni Node.js sono scritte in puro JavaScript e possono essere eseguite all'interno dell'ambiente Node.js su Windows, Linux, ecc ...

Versioni

Versione	Data di rilascio
V8.2.1	2017/07/20
v8.2.0	2017/07/19
v8.1.4	2017/07/11
v8.1.3	2017/06/29
V8.1.2	2017/06/15
v8.1.1	2017/06/13
v8.1.0	2017/06/08
v8.0.0	2017/05/30
v7.10.0	2017/05/02
v7.9.0	2017/04/11
v7.8.0	2017/03/29
v7.7.4	2017/03/21
v7.7.3	2017/03/14
v7.7.2	2017/03/08
v7.7.1	2017/03/02
v7.7.0	2017/02/28

Versione	Data di rilascio
V7.6.0	2017/02/21
v7.5.0	2017/01/31
v7.4.0	2017/01/04
v7.3.0	2016/12/20
v7.2.1	2016/12/06
v7.2.0	2016/11/22
v7.1.0	2016/11/08
v7.0.0	2016/10/25
v6.11.0	2017/06/06
v6.10.3	2017/05/02
v6.10.2	2017/04/04
v6.10.1	2017/03/21
v6.10.0	2017/02/21
v6.9.5	2017/01/31
v6.9.4	2017/01/05
v6.9.3	2017/01/05
v6.9.2	2016/12/06
v6.9.1	2016/10/19
v6.9.0	2016/10/18
v6.8.1	2016/10/14
v6.8.0	2016/10/12
v6.7.0	2016/09/27
v6.6.0	2016/09/14
v6.5.0	2016/08/26
v6.4.0	2016/08/12

Versione	Data di rilascio
v6.3.1	2016/07/21
v6.3.0	2016/07/06
V6.2.2	2016/06/16
V6.2.1	2016/06/02
V6.2.0	2016/05/17
V6.1.0	2016/05/05
v6.0.0	2016/04/26
v5.12.0	2016/06/23
v5.11.1	2016/05/05
v5.11.0	2016/04/21
v5.10.1	2016/04/05
v5.10	2016/04/01
v5.9	2016/03/16
v5.8	2016/03/09
quella 5.7	2016/02/23
v5.6	2016/02/09
v5.5	2016/01/21
V5.4	2016/01/06
V5.3	2015/12/15
v5.2	2015/12/09
v5.1	2015/11/17
v5.0	2015/10/29
v4.4	2016/03/08
V4.3	2016/02/09
v4.2	2015/10/12

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

Versione	Data di rilascio
v0.8	2012-06-22
v0.7	2012-01-17
v0.6	2011-11-04
v0.5	2011-08-26
v0.4	2011-08-26
v0.3	2011-08-26
v0.2	2011-08-26
v0.1	2011-08-26

Examples

Ciao server World HTTP

Innanzitutto, installa Node.js per la tua piattaforma.

In questo esempio creeremo un server HTTP in ascolto sulla porta 1337, che invia `Hello, World!` al browser. Si noti che, invece di utilizzare la porta 1337, è possibile utilizzare qualsiasi numero di porta a scelta che al momento non è utilizzato da nessun altro servizio.

Il modulo `http` è un ***modulo core*** Node.js (un modulo incluso nell'origine di Node.js, che non richiede l'installazione di risorse aggiuntive). Il modulo `http` fornisce la funzionalità per creare un server HTTP utilizzando il metodo `http.createServer()` . Per creare l'applicazione, creare un file contenente il seguente codice JavaScript.

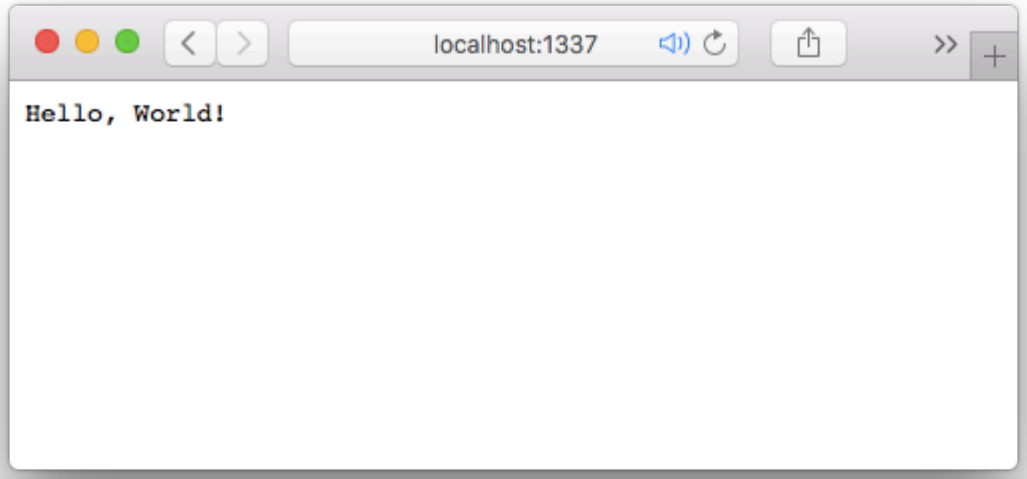
```
const http = require('http'); // Loads the http module

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

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

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

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

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

Salva il file con qualsiasi nome di file. In questo caso, se lo `hello.js` possiamo eseguire l'applicazione andando nella directory in cui si trova il file e usando il seguente comando:

```
node hello.js
```

È possibile accedere al server creato con l'URL http: // localhost: 1337 o http://127.0.0.1:1337 nel browser.

Una pagina web semplice apparirà con un testo "Hello, World!" In alto, come mostrato nello screenshot qui sotto.



Esempio online modificabile.

Ciao linea di comando mondiale

Node.js può anche essere utilizzato per creare utilità da riga di comando. L'esempio seguente legge il primo argomento dalla riga di comando e stampa un messaggio Hello.

Per eseguire questo codice su un sistema Unix:

1. Crea un nuovo file e incolla il codice qui sotto. Il nome file è irrilevante.
2. Rendi questo file eseguibile con `chmod 700 FILE_NAME`
3. Esegui l'app con `./APP_NAME David`

Su Windows fai il passaggio 1 ed `node APP_NAME David` con il `node APP_NAME David`

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

'use strict';

/*
    The command line arguments are stored in the `process.argv` array,
    which has the following structure:
    [0] The path of the executable that started the Node.js process
    [1] The path to this application
    [2-n] the command line arguments
```

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

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

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

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

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

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

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

Installazione ed esecuzione di Node.js

Per iniziare, installa Node.js sul tuo computer di sviluppo.

Windows: vai alla pagina di download e scarica / esegui il programma di installazione.

Mac: vai alla pagina di download e scarica / esegui il programma di installazione. In alternativa, è possibile installare il nodo tramite Homebrew utilizzando il `brew install node` . Homebrew è un pacchetto di comandi a riga di comando per Macintosh, e ulteriori informazioni a riguardo sono disponibili sul sito Web di Homebrew .

Linux: seguire le istruzioni per la distro sulla pagina di installazione della riga di comando .

Esecuzione di un programma di nodo

Per eseguire un programma Node.js, esegui semplicemente il `node app.js` o `nodejs app.js` , dove `app.js` è il nome file del tuo codice sorgente dell'app nodo. Non è necessario includere il suffisso `.js` per il nodo per trovare lo script che desideri eseguire.

In alternativa, con i sistemi operativi basati su UNIX, un programma Node può essere eseguito come uno script terminale. Per fare ciò, è necessario iniziare con uno shebang che punta all'interprete Node, come `#!/usr/bin/env node` . Il file deve anche essere impostato come eseguibile, che può essere fatto usando `chmod` . Ora lo script può essere eseguito direttamente dalla riga di comando.

Distribuzione dell'applicazione online

Quando si distribuisce l'app in un ambiente ospitato (specifico per Node.js), in genere questo ambiente offre una variabile di ambiente `PORT` che è possibile utilizzare per eseguire il server. La modifica del numero di porta in `process.env.PORT` consente di accedere all'applicazione.

Per esempio,

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

Inoltre, se desideri accedere a questo offline durante il debug, puoi utilizzare questo:

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

dove `3000` è il numero di porta offline.

Debug della tua applicazione NodeJS

È possibile utilizzare il nodo-inspector. Esegui questo comando per installarlo tramite npm:

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

Quindi puoi eseguire il debug dell'applicazione usando

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

Il repository Github può essere trovato qui: https://github.com/node-inspector/node-inspector

---

Debugging nativamente

Puoi anche eseguire il debug di node.js in modo nativo avviandolo in questo modo:

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

Per interrompere il debugger esattamente in una riga di codice che desideri, usa questo:

```
debugger;
```

Per maggiori informazioni vedi qui .

In node.js 8 usa il seguente comando:

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

Quindi apri `about://inspect` in una versione recente di Google Chrome e seleziona lo script del nodo per ottenere l'esperienza di debug di Chrome's DevTools.

Ciao mondo con Express

Nell'esempio seguente viene utilizzato Express per creare un server HTTP in ascolto sulla porta 3000, che risponde con "Hello, World!". Express è un framework web comunemente utilizzato che è utile per creare API HTTP.

Innanzitutto, crea una nuova cartella, ad esempio `myApp` . Vai su `myApp` e crea un nuovo file JavaScript contenente il seguente codice (`hello.js` per esempio). Quindi installare il modulo express usando `npm install --save express` dalla riga di comando. *Fare riferimento a questa documentazione per ulteriori informazioni su come installare i pacchetti* .

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

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

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

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

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

Dalla riga di comando, eseguire il seguente comando:

```
node hello.js
```

Apri il browser e vai a `http://localhost:3000` o `http://127.0.0.1:3000` per vedere la risposta.

Per ulteriori informazioni sul framework Express, è possibile controllare la sezione Web Apps con Express

Hello World routing di base

Una volta compreso come creare un server HTTP con nodo, è importante capire come farlo "fare" le cose in base al percorso a cui un utente ha navigato. Questo fenomeno è chiamato "routing".

L'esempio più semplice di questo sarebbe verificare `if (request.url === 'some/path/here')` , e quindi chiamare una funzione che risponde con un nuovo file.

Un esempio di questo può essere visto qui:

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

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

http.createServer(function (request, response) {

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

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

}).listen(1337);
```

Se continui a definire le tue "rotte" come questa, però, finirai con una funzione di callback enorme, e non vogliamo un pasticcio gigante come quello, quindi vediamo se riusciamo a ripulire tutto.

Per prima cosa, memorizziamo tutti i nostri percorsi in un oggetto:

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

Ora che abbiamo memorizzato 2 percorsi in un oggetto, ora possiamo controllarli nel nostro callback principale:

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

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

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

}).listen(1337);
```

Ora ogni volta che provi a navigare nel tuo sito web, controllerà l'esistenza di quel percorso nei tuoi percorsi e chiamerà la rispettiva funzione. Se non viene trovata alcuna route, il server risponderà con un 404 (non trovato).

Ed ecco fatto: il routing con l'API di HTTP Server è molto semplice.

Le uniche differenze principali tra questa e una normale connessione TCP sono la chiave privata e il certificato pubblico che dovrai impostare in un oggetto opzione.

Come creare una chiave e un certificato

Il primo passo in questo processo di sicurezza è la creazione di una chiave privata. E qual è questa chiave privata? Fondamentalmente, è un insieme di rumore casuale che viene utilizzato per crittografare le informazioni. In teoria, potresti creare una chiave e usarla per crittografare quello che vuoi. Ma è una buona pratica avere diverse chiavi per cose specifiche. Perché se qualcuno ruba la tua chiave privata, è come se qualcuno rubasse le chiavi della tua casa. Immagina se hai usato la stessa chiave per bloccare la tua auto, garage, ufficio, ecc.

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

Una volta che abbiamo la nostra chiave privata, possiamo creare una CSR (richiesta di firma del certificato), che è la nostra richiesta di far firmare la chiave privata da un'autorità immaginaria. Questo è il motivo per cui devi inserire le informazioni relative alla tua azienda. Queste informazioni verranno visualizzate dall'autorità di firma e utilizzate per verificare l'utente. Nel nostro caso, non importa ciò che scrivi, dal momento che nel prossimo passaggio firmeremo noi stessi il nostro certificato.

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

Ora che abbiamo completato il nostro lavoro cartaceo, è tempo di fingere di essere un'autentica autorità firmataria.

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

Ora che hai la chiave privata e il certificato pubblico, puoi stabilire una connessione sicura tra due app NodeJS. E, come puoi vedere nel codice di esempio, è un processo molto semplice.

Importante!

Dal momento che abbiamo creato noi stessi il certificato pubblico, in tutta onestà, il nostro certificato non ha valore, perché siamo nomini. Il server NodeJS non si fiderà di tale certificato per impostazione predefinita, ed è per questo motivo che è necessario dirlo per fidarsi effettivamente del nostro cert con la seguente opzione rejectUnauthorized: false. **Molto importante** : non impostare mai questa variabile su true in un ambiente di produzione.

TLS Socket Server

```
'use strict';
```

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

const PORT = 1337;
const HOST = '127.0.0.1'

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

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

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

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

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

    });

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

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

    });

});

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

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

});

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

    console.error(error);

    // Close the connection after the error occurred.
    server.destroy();

});
```

TLS Socket Client

```
'use strict';

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

```
const PORT = 1337;
const HOST = '127.0.0.1'

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

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

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

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

});

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

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

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

});

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

    console.log("Connection closed");

});

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

    console.error(error);

    // Close the connection after the error occurred.
    client.destroy();

});
```

Ciao mondo nella REPL

Quando chiamato senza argomenti, Node.js avvia un REPL (Read-Eval-Print-Loop) noto anche come " *Node shell* ".

Al prompt dei comandi digitare `node` .

```
$ node
>
```

Al prompt della shell Nodo `>` tipo "Ciao Mondo!"

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

Moduli core

Node.js è un motore Javascript (motore V8 di Google per Chrome, scritto in C ++) che consente di eseguire Javascript all'esterno del browser. Mentre numerose librerie sono disponibili per estendere le funzionalità di Node, il motore viene fornito con una serie di *moduli principali che* implementano funzionalità di base.

Ci sono attualmente 34 moduli principali inclusi nel nodo:

```
[ 'assert',
  'buffer',
  'c/c++_addons',
  'child_process',
  'cluster',
  'console',
  'crypto',
  'deprecated_apis',
  'dns',
  'domain',
  'Events',
  'fs',
  'http',
  'https',
  'module',
  'net',
  'os',
  'path',
  'punycode',
  'querystring',
  'readline',
  'repl',
  'stream',
  'string_decoder',
  'timers',
  'tls_(ssl)',
  'tracing',
  'tty',
  'dgram',
  'url',
  'util',
  'v8',
  'vm',
  'zlib' ]
```

Questo elenco è stato ottenuto dall'API di documentazione del nodo https://nodejs.org/api/all.html (file JSON: https://nodejs.org/api/all.json).

Tutti i moduli di base a colpo d'occhio

affermare

Il modulo `assert` fornisce una semplice serie di test di asserzione che possono essere utilizzati per testare gli invarianti.

buffer

Prima dell'introduzione di `TypedArray` in ECMAScript 2015 (ES6), il linguaggio JavaScript non aveva alcun meccanismo per leggere o manipolare flussi di dati binari. La classe `Buffer` è stata introdotta come parte dell'API Node.js per rendere possibile l'interazione con i flussi di ottetti nel contesto di operazioni come flussi TCP e operazioni del file system.

Ora che `TypedArray` è stato aggiunto in ES6, la classe `Buffer` implementa l'API `Uin t8Array` in un modo più ottimizzato e adatto ai casi d'uso di Node.js.

C / C ++ _ addons

I componenti aggiuntivi di Node.js sono oggetti condivisi collegati dinamicamente, scritti in C o C ++, che possono essere caricati in Node.js usando la funzione `require()` e usati come se fossero un normale modulo Node.js. Vengono utilizzati principalmente per fornire un'interfaccia tra JavaScript in esecuzione nelle librerie Node.js e C / C ++.

child_process

Il modulo `child_process` fornisce la possibilità di generare processi figlio in un modo simile, ma non identico, a popen (3).

grappolo

Una singola istanza di Node.js viene eseguita in un singolo thread. Per sfruttare i sistemi multi-core, l'utente a volte desidera avviare un cluster di processi Node.js per gestire il carico. Il modulo cluster consente di creare facilmente processi figlio che condividono tutte le porte del server.

consolle

Il modulo `console` fornisce una semplice console di debug che è simile al meccanismo della console JavaScript fornito dai browser web.

crypto

Il modulo `crypto` fornisce funzionalità crittografiche che include un set di wrapper per le funzioni di hash, HMAC, cifratura, decifratura, firma e verifica di OpenSSL.

deprecated_apis

Node.js può deprecare API quando: (a) l'utilizzo dell'API è considerato non sicuro, (b) è stata resa disponibile un'API alternativa migliorata oppure (c) sono previste interruzioni delle modifiche

all'API in una futura versione principale .

dns

Il modulo `dns` contiene funzioni appartenenti a due diverse categorie:

1. Funzioni che utilizzano le funzionalità del sistema operativo sottostante per eseguire la risoluzione dei nomi e che non eseguono necessariamente alcuna comunicazione di rete. Questa categoria contiene solo una funzione: `dns.lookup()` .
2. Funzioni che si connettono a un server DNS effettivo per eseguire la risoluzione dei nomi e che utilizzano *sempre* la rete per eseguire query DNS. Questa categoria contiene tutte le funzioni nel modulo `dns` *tranne* `dns.lookup()` .

dominio

Questo modulo è in attesa di deprecazione . Una volta completata l'API di sostituzione, questo modulo sarà completamente deprecato. La maggior parte degli utenti **non** dovrebbe avere motivo di usare questo modulo. Gli utenti che devono assolutamente disporre delle funzionalità fornite dai domini possono affidarsi al momento, ma in futuro dovrebbero aspettarsi di dover eseguire la migrazione a una soluzione diversa.

eventi

Gran parte dell'API core di Node.js è basata su un'architettura asincrona basata sugli eventi asincrona in cui determinati tipi di oggetti (chiamati "emettitori") emettono periodicamente eventi denominati che causano la chiamata di oggetti Function ("listener").

fs

I / O di file sono forniti da semplici wrapper attorno alle funzioni POSIX standard. Per utilizzare questo modulo è `require('fs')` . Tutti i metodi hanno forme asincrone e sincrone.

http

Le interfacce HTTP in Node.js sono progettate per supportare molte funzionalità del protocollo che sono state tradizionalmente difficili da utilizzare. In particolare, messaggi di grandi dimensioni, possibilmente codificati per il chunk. L'interfaccia fa attenzione a non bufferizzare intere richieste o risposte - l'utente è in grado di trasmettere dati.

https

HTTPS è il protocollo HTTP su TLS / SSL. In Node.js questo è implementato come un modulo separato.

modulo

Node.js ha un semplice sistema di caricamento dei moduli. In Node.js, i file e i moduli sono in corrispondenza uno a uno (ogni file viene trattato come un modulo separato).

netto

Il modulo `net` fornisce un wrapper di rete asincrono. Contiene funzioni per la creazione di server e client (chiamati flussi). Puoi includere questo modulo con `require('net');` .

os

Il modulo `os` fornisce numerosi metodi di utilità relativi al sistema operativo.

sentiero

Il modulo `path` fornisce utility per lavorare con i percorsi di file e directory.

Punycode

La versione del modulo punycode in bundle in Node.js è deprecata .

stringa della domanda

Il modulo `querystring` fornisce utility per l'analisi e la formattazione delle stringhe di query URL.

linea di lettura

Il modulo `readline` fornisce un'interfaccia per leggere i dati da un flusso leggibile (come `process.stdin`) una riga alla volta.

repl

Il modulo `repl` fornisce un'implementazione Read-Eval-Print-Loop (REPL) che è disponibile sia come programma standalone che includibile in altre applicazioni.

ruscello

Un flusso è un'interfaccia astratta per lavorare con i dati di streaming in Node.js. Il modulo `stream` fornisce un'API di base che semplifica la creazione di oggetti che implementano l'interfaccia di streaming.

Esistono molti oggetti di flusso forniti da Node.js. Ad esempio, una richiesta a un server HTTP e `process.stdout` sono entrambe istanze di flusso.

string_decoder

Il modulo `string_decoder` fornisce un'API per decodificare gli oggetti `Buffer` in stringhe in modo da preservare i caratteri codificati UTF-8 e UTF-16 codificati.

temporizzatori

Il modulo `timer` espone un'API globale per le funzioni di pianificazione da richiamare in un determinato periodo di tempo. Poiché le funzioni del timer sono globali, non è necessario chiamare `require('timers')` per utilizzare l'API.

Le funzioni del timer all'interno di Node.js implementano un'API simile all'API dei timer fornita dai browser Web, ma utilizzano un'implementazione interna diversa costruita attorno al Loop eventi di

Node.js.

tls_ (SSL)

Il modulo `tls` fornisce un'implementazione dei protocolli Transport Layer Security (TLS) e Secure Socket Layer (SSL) costruiti su OpenSSL.

tracciato

Trace Event fornisce un meccanismo per centralizzare le informazioni di traccia generate da V8, nucleo del nodo e codice dello spazio utente.

La traccia può essere abilitata passando il `--trace-events-enabled` all'avvio di un'applicazione Node.js.

tty

Il modulo `tty` fornisce le classi `tty.ReadStream` e `tty.WriteStream` . Nella maggior parte dei casi, non sarà necessario o possibile utilizzare direttamente questo modulo.

dgram

Il modulo `dgram` fornisce un'implementazione di socket UDP Datagram.

url

Il modulo `url` fornisce utility per la risoluzione e l'analisi dell'URL.

util

Il modulo `util` è progettato principalmente per supportare le esigenze delle API interne di Node.js. Tuttavia, molte delle utilità sono utili anche per gli sviluppatori di applicazioni e moduli.

v8

Il modulo `v8` espone le API specifiche per la versione di V8 integrata nel binario Node.js.

Nota : le API e l'implementazione sono soggette a modifiche in qualsiasi momento.

vm

Il modulo `vm` fornisce API per la compilazione e l'esecuzione del codice all'interno dei contesti della macchina virtuale V8. Il codice JavaScript può essere compilato ed eseguito immediatamente o compilato, salvato ed eseguito successivamente.

Nota : il modulo vm non è un meccanismo di sicurezza. ***Non usarlo per eseguire codice non affidabile*** .

zlib

Il modulo `zlib` fornisce funzionalità di compressione implementate usando Gzip e Deflate / Inflate.

Come ottenere un server Web HTTPS di base attivo e funzionante!

Una volta che node.js è installato sul tuo sistema, puoi semplicemente seguire la procedura seguente per ottenere un server Web di base in esecuzione con supporto per HTTP e HTTPS!

Passaggio 1: creare un'autorità di certificazione

1. creare la cartella in cui si desidera memorizzare la chiave e il certificato:

```
mkdir conf
```

---

2. vai a quella directory:

```
cd conf
```

---

3. prendi questo file `ca.cnf` da usare come scorciatoia di configurazione:

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

---

4. creare una nuova autorità di certificazione utilizzando questa configurazione:

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

---

5. ora che abbiamo la nostra autorità di certificazione in `ca-key.pem` e `ca-cert.pem` , `ca-key.pem` una chiave privata per il server:

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

---

6. prendi questo file `server.cnf` da usare come scorciatoia di configurazione:

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

---

7. generare la richiesta di firma del certificato utilizzando questa configurazione:

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

---

8. firmare la richiesta:

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

Passaggio 2: installa il certificato come certificato di origine

1. copia il tuo certificato nella cartella dei certificati di root:

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

---

2. aggiornare l'archivio di CA:

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

Passaggio 3: avvio del server nodo

Innanzitutto, si desidera creare un file `server.js` che contenga il proprio codice server attuale.

L'installazione minima per un server HTTPS in Node.js sarebbe qualcosa del genere:

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

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

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

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

Se vuoi anche supportare le richieste http, devi fare solo questa piccola modifica:

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

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

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

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

1. vai alla directory in cui si trova il tuo `server.js` :

   ```
   cd /path/to
   ```

---

2. eseguire `server.js` :

```
node server.js
```

Leggi Iniziare con Node.js online: https://riptutorial.com/it/node-js/topic/340/iniziare-con-node-js

Capitolo 2: Ambiente

Examples

Accesso alle variabili d'ambiente

La proprietà `process.env` restituisce un oggetto contenente l'ambiente utente.

Restituisce un oggetto come questo:

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

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

Se imposti la variabile di ambiente `FOO` su `foobar` , sarà accessibile con:

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

argomenti della riga di comando process.argv

process.argv è una matrice contenente gli argomenti della riga di comando. Il primo elemento sarà `node` , il secondo elemento sarà il nome del file JavaScript. Gli elementi successivi saranno gli argomenti aggiuntivi della riga di comando.

Esempio di codice:

Output sum di tutti gli argomenti della riga di comando

`index.js`

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

console.log(sum);
```

Uso Exaple:

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

L'uscita sarà `20`

Una breve spiegazione del codice:

Qui in for loop `for (i = 2; i < process.argv.length; i++)` loop inizia con 2 perché i primi due elementi nell'array process.argv sono **sempre** `['path/to/node.exe', 'path/to/js/file', ...]`

Conversione in numero `Number(process.argv[i])` perché gli elementi nell'array process.argv sono **sempre** stringhe

Utilizzo di diverse proprietà / configurazione per ambienti diversi come dev, qa, staging, ecc.

Le applicazioni su larga scala spesso hanno bisogno di proprietà diverse quando funzionano su ambienti diversi. possiamo ottenere ciò passando argomenti all'applicazione NodeJs e utilizzando lo stesso argomento nel processo del nodo per caricare un file di proprietà dell'ambiente specifico.

Supponiamo di avere due file di proprietà per diversi ambienti.

- dev.json

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

- qa.json

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

Il codice seguente nell'applicazione esporterà il rispettivo file di proprietà che vogliamo utilizzare.

Supponiamo che il codice sia in environment.js

```
process.argv.forEach(function (val, index, array) {
```

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

Diamo argomenti all'applicazione come seguendo

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

se usiamo process manager come *per sempre* che sia semplice come

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

Come utilizzare il file di configurazione

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

Caricamento delle proprietà dell'ambiente da un "file di proprietà"

- Installa il lettore di proprietà:

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

- Crea una **directory env** per memorizzare i tuoi file di proprietà:

```
mkdir env
```

- Crea **environments.js** :

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

- Esempio di file di proprietà **development.properties** :

```
# Dev properties
[main]
# Application port to run the node server
app.port=8080

[database]
```

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

- Esempio di utilizzo delle proprietà caricate:

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

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

- Avvio del server Express

```
npm start env=development
```

o

```
npm start env=production
```

Leggi Ambiente online: https://riptutorial.com/it/node-js/topic/2340/ambiente

Capitolo 3: Analizzare gli argomenti della riga di comando

Examples

Passando azione (verbo) e valori

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

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

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

options.parse(process.argv);

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

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

Passare gli switch booleani

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

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

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

Leggi Analizzare gli argomenti della riga di comando online: https://riptutorial.com/it/node-js/topic/6174/analizzare-gli-argomenti-della-riga-di-comando

Capitolo 4: API CRUD basata su REST semplice

Examples

API REST per CRUD in Express 3+

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

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

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

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

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

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

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

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

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

Leggi API CRUD basata su REST semplice online: https://riptutorial.com/it/node-js/topic/5850/api-crud-basata-su-rest-semplice

Capitolo 5: App Web con Express

introduzione

Express è un framework di applicazioni web Node.js minimale e flessibile, che fornisce un robusto set di funzionalità per la creazione di applicazioni Web.

Il sito web ufficiale di Express è expressjs.com . La fonte può essere trovata su GitHub .

Sintassi

- app.get (percorso [, middleware], callback [, callback ...])
- app.put (percorso [, middleware], callback [, callback ...])
- app.post (percorso [, middleware], callback [, callback ...])
- app ['delete'] (percorso [, middleware], callback [, callback ...])
- app.use (percorso [, middleware], callback [, callback ...])
- app.use (callback)

Parametri

Parametro	Dettagli
`path`	Specifica la parte del percorso o l'URL che verrà gestito dal callback specificato.
`middleware`	Una o più funzioni che verranno chiamate prima della richiamata. Essenzialmente un concatenamento di più funzioni di `callback` . Utile per una gestione più specifica, ad esempio l'autorizzazione o la gestione degli errori.
`callback`	Una funzione che verrà utilizzata per gestire le richieste nel `path` specificato. Sarà chiamato come `callback(request, response, next)` , dove `request` , `response` e `next` sono descritti di seguito.
`request` *richiamata*	Un oggetto che incapsula i dettagli sulla richiesta HTTP che il callback è stato chiamato a gestire.
`response`	Un oggetto che viene utilizzato per specificare in che modo il server deve rispondere alla richiesta.
`next`	Un callback che passa il controllo sulla prossima route corrispondente. Accetta un oggetto errore opzionale.

Examples

Iniziare

Dovrai prima creare una directory, accedervi nella tua shell e installare Express usando npm eseguendo `npm install express --save`

Crea un file e `app.js` e aggiungi il codice seguente che crea un nuovo server Express e aggiunge un endpoint (`/ping`) con il metodo `app.get` :

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

const app = express();

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

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

Per eseguire il tuo script usa il seguente comando nella tua shell:

```
> node app.js
```

La tua applicazione accetterà le connessioni sulla porta localhost 8080. Se l'argomento nome host per `app.listen` viene omesso, il server accetterà le connessioni sull'indirizzo IP della macchina e su localhost. Se il valore della porta è 0, il sistema operativo assegnerà una porta disponibile.

Una volta che lo script è in esecuzione, puoi testarlo in una shell per confermare che ottieni la risposta prevista, "pong", dal server:

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

È anche possibile aprire un browser Web, accedere all'URL http: // localhost: 8080 / ping per visualizzare l'output

Routing di base

Per prima cosa crea un'app express:

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

Quindi puoi definire percorsi come questo:

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

Questa struttura funziona per tutti i metodi HTTP e prevede un percorso come primo argomento e un gestore per quel percorso, che riceve gli oggetti richiesta e risposta. Quindi, per i metodi HTTP di base, questi sono i percorsi

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

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

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

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

Puoi controllare l'elenco completo dei verbi supportati qui . Se si desidera definire lo stesso comportamento per una route e tutti i metodi HTTP, è possibile utilizzare:

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

o

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

o

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

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

È possibile concatenare le definizioni del percorso per un singolo percorso

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

È inoltre possibile aggiungere funzioni a qualsiasi metodo HTTP. Eseguiranno prima del callback finale e prenderanno come parametri i parametri (req, res, next).

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

Le tue callback finali possono essere archiviate in un file esterno per evitare di inserire troppo codice in un file:

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

E poi nel file contenente i tuoi percorsi:

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

Questo renderà il tuo codice molto più pulito.

Ottenere informazioni dalla richiesta

Per ottenere informazioni dall'URL richiedente (notare che `req` è l'oggetto richiesta nella funzione di gestione dei percorsi). Considera questa definizione di percorso **`/settings/:user_id`** e questo particolare esempio **`/settings/32135?field=name`**

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

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

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

Puoi anche ottenere intestazioni della richiesta, come questa

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

Per semplificare l'acquisizione di altre informazioni puoi usare i middleware. Ad esempio, per ottenere le informazioni sulla body della richiesta, è possibile utilizzare il middleware body-parser , che trasformerà il corpo della richiesta grezza in formato utilizzabile.

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

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

Supponiamo ora una richiesta come questa

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

Puoi accedere al nome postato come questo

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

In modo simile, puoi accedere ai cookie dalla richiesta, hai anche bisogno di un middleware come cookie-parser

```
req.cookies.name
```

Applicazione espressa modulare

Per rendere le fabbriche di router di applicazioni modulari per uso express:

Modulo:

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

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

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

    return router;
};
```

Applicazione:

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

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

Ciò renderà la tua applicazione modulare, personalizzabile e il tuo codice riutilizzabile.

Quando si accede a `http://<hostname>:8080/api/v1/greet` l'output sarà `Hello world`

Esempio più complicato

Esempio con servizi che mostrano vantaggi di fabbrica del middleware.

Modulo:

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

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

    router.get('/greet', (req, res, next) => {
        res.end(
            service.createGreeting(req.query.name || 'Stranger')
        );
    });
```

```
    return router;
};
```

Applicazione:

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

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

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

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

Quando si accede a `http://<hostname>:8080/api/v1/service1/greet?name=World` l'output sarà `Hello, World` e accederà a `http://<hostname>:8080/api/v1/service2/greet?name=World` l'output sarà `Hi, World` .

Utilizzo di un motore di modelli

Utilizzo di un motore di modelli

Il codice seguente configurerà Jade come motore di template. (Nota: Jade è stata ribattezzata come `pug` a dicembre 2015.)

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

const PORT = 3000; //Randomly chosen port

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

//Creates a Root Route
app.get('/',function(req, res){
    res.render('index');  //renders the index.jade file into html and returns as a response.
The render function optionally takes the data to pass to the view.
});

//Starts the Express server with a callback
```

```
app.listen(PORT, function(err) {
    if (!err) {
        console.log('Server is running at port', PORT);
    } else {
        console.log(JSON.stringify(err));
    }
});
```

Allo stesso modo, potrebbero essere utilizzati anche altri modelli di template come `Handlebars` (`hbs`) o `ejs` . Ricorda di `npm install` il motore di template. Per Handlebars utilizziamo il pacchetto `hbs` , per Jade abbiamo un pacchetto `jade` e per EJS abbiamo un pacchetto `ejs` .

Esempio di modello EJS

Con EJS (come altri modelli express), puoi eseguire il codice del server e accedere alle tue variabili del server dal tuo HTML.
In EJS è fatto usando " `<%` " come tag di inizio e " `%>` " come tag di fine, le variabili passate come parametri di rendering sono accessibili usando `<%=var_name%>`
Ad esempio, se si dispone di array di forniture nel codice del server
puoi ricollegarlo usando

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

Come puoi vedere nell'esempio ogni volta che cambi codice server e HTML devi chiudere il tag EJS corrente e aprirne uno successivo, qui abbiamo voluto creare `li` dentro il comando `for` quindi abbiamo dovuto chiudere il nostro tag EJS alla fine del `for` e crea un nuovo tag solo per le parentesi graffe
un altro esempio
se vogliamo inserire la versione predefinita di input come variabile dal lato server, usiamo `<%=` per esempio:

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

Qui la variabile del messaggio passata dal lato server sarà il valore predefinito del tuo input, tieni presente che se non hai passato la variabile del messaggio dal lato server, EJS genererà un'eccezione. Puoi passare i parametri usando `res.render('index', {message: message});` (per il file ejs chiamato index.ejs).

Nei tag EJS puoi anche utilizzare `if` , `while` o qualsiasi altro comando javascript che desideri.

API JSON con ExpressJS

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

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

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

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

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

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

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

Su `http://localhost:8080/` oggetto di output

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

Servire file statici

Quando si crea un server Web con Express, è spesso richiesto di offrire una combinazione di contenuti dinamici e file statici.

Ad esempio, potresti avere index.html e script.js che sono file statici mantenuti nel file system.

È comune utilizzare la cartella denominata "public" per avere file statici. In questo caso la struttura della cartella potrebbe essere simile a:

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

Ecco come configurare Express per servire file statici:

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

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

Nota: una volta configurata la cartella, index.html, script.js e tutti i file nella cartella "pubblica" saranno disponibili nel percorso principale (non è necessario specificare `/public/` nell'URL). Questo perché, express cerca i file relativi alla cartella statica configurata. È possibile specificare *il prefisso del percorso virtuale* come mostrato di seguito:

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

renderà le risorse disponibili sotto `/static/` prefisso.

Più cartelle

È possibile definire più cartelle contemporaneamente:

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

Quando serve le risorse Express esaminerà la cartella nell'ordine di definizione. In caso di file con lo stesso nome, verrà pubblicato quello nella prima cartella di corrispondenza.

Percorsi con nomi in stile Django

Un grosso problema è che le rotte con nome prezioso non sono supportate da Express out of the box. La soluzione è installare il pacchetto di terze parti supportato, ad esempio express-reverse :

```
npm install express-reverse
```

Collegalo al tuo progetto:

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

Quindi usarlo come:

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

Lo svantaggio di questo approccio è che non è possibile utilizzare il modulo Express `route` come mostrato nell'utilizzo avanzato del router . La soluzione alternativa è passare la tua `app` come parametro alla tua fabbrica di router:

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

E usalo come:

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

Da ora in poi puoi capire come definire le funzioni per unirle con spazi dei nomi personalizzati specificati e puntare ai controller appropriati.

Gestione degli errori

Gestione degli errori di base

Per impostazione predefinita, Express cercherà una vista "errore" nella directory `/views` per il rendering. Basta creare la vista 'errore' e posizionarla nella directory views per gestire gli errori. Gli errori vengono scritti con il messaggio di errore, lo stato e la traccia dello stack, ad esempio:

views / error.pug

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

Gestione avanzata degli errori

Definisci le tue funzioni middleware di gestione degli errori alla fine dello stack di funzioni middleware. Questi hanno quattro argomenti invece di tre `(err, req, res, next)` per esempio:

app.js

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

    //pass error to the next matching route.
    next(err);
});

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

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

```
});
```

È possibile definire diverse funzioni middleware di gestione degli errori, proprio come si farebbe con le normali funzioni middleware.

Utilizzo del middleware e della prossima richiamata

Express passa una `next` richiamata a tutte le funzioni di gestore di route e middleware che possono essere utilizzate per interrompere la logica per rotte singole su più gestori. La chiamata a `next()` senza argomenti indica a express di continuare con il prossimo middleware o gestore di route corrispondente. Se si chiama `next(err)` con un errore, verrà attivato qualsiasi middleware del gestore degli errori. La chiamata `next('route')` ignorerà qualsiasi middleware successivo sulla rotta corrente e passerà alla successiva route corrispondente. Ciò consente di disaccoppiare la logica del dominio in componenti riutilizzabili che sono autonomi, più semplici da testare e più facili da mantenere e modificare.

Più percorsi di corrispondenza

Le richieste a `/api/foo` o a `/api/bar` eseguiranno il gestore iniziale per cercare il membro e quindi passare il controllo al gestore effettivo per ogni percorso.

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

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

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

Gestore degli errori

I gestori degli errori sono middleware con la `function(err, req, res, next)` firma `function(err, req, res, next)` . Possono essere impostati per percorso (ad esempio `app.get('/foo', function(err, req, res, next)`) ma in genere, un singolo gestore di errori che esegue il rendering di una pagina di errore è sufficiente.

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

```
});

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

middleware

Ciascuna delle funzioni di cui sopra è in realtà una funzione middleware che viene eseguita ogni volta che una richiesta corrisponde alla rotta definita, ma qualsiasi numero di funzioni middleware può essere definito su una singola rotta. Ciò consente di definire il middleware in file separati e la logica comune da riutilizzare su più percorsi.

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

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

In questo esempio, ciascuna funzione middleware potrebbe trovarsi nel proprio file o in una variabile altrove nel file in modo che possa essere riutilizzata in altre route.

Gestione degli errori

I documenti di base possono essere trovati **qui**

```
app.get('/path/:id(\\d+)', function (req, res, next) { // please note: "next" is passed
    if (req.params.id == 0) // validate param
```

```
        return next(new Error('Id is 0')); // go to first Error handler, see below

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

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

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

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

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

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

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

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

Appendice A

```
// "In Express, 404 responses are not the result of an error,
// so the error-handler middleware will not capture them."
// You can change it.
app.use(function(req, res, next) {
    next(new Error(404));
});
```

Appendice B

```
// How to define custom error
```

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

Hook: come eseguire il codice prima di qualsiasi req e dopo qualsiasi res

`app.use()` e middleware possono essere utilizzati per "before" e una combinazione di eventi close e finish può essere utilizzata per "after".

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

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

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

Un esempio di questo è il middleware del logger , che verrà aggiunto al log dopo la risposta per impostazione predefinita.

Assicurati che questo "middleware" sia utilizzato prima di `app.router` quanto l'ordine è importante.

Il post originale è **qui**

Gestire richieste POST

Proprio come gestisci le richieste get in Express con il metodo app.get, puoi utilizzare il metodo app.post per gestire le richieste post.

Ma prima di poter gestire le richieste POST, sarà necessario utilizzare il middleware `body-parser` . Semplicemente analizza il corpo di `POST` , `PUT` , `DELETE` e altre richieste.

`Body-Parser` middleware `Body-Parser` analizza il corpo della richiesta e lo trasforma in un oggetto disponibile in `req.body`

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

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

const app = express();

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

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

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

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

});

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

Impostazione dei cookie con cookie-parser

Di seguito è riportato un esempio per l'impostazione e la lettura dei cookie utilizzando il modulo cookie-parser :

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

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

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

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

app.listen(3000);
```

Middleware personalizzato in Express

In Express, è possibile definire i middleware che possono essere utilizzati per controllare le richieste o impostare alcune intestazioni in risposta.

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

Esempio

Il codice seguente aggiunge l' `user` all'oggetto richiesta e passa il controllo alla successiva route

corrispondente.

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

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

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

app.listen(3000);
```

Gestione degli errori in Express

In Express, è possibile definire un gestore di errori unificato per la gestione degli errori verificatisi nell'applicazione. Definire quindi il gestore alla fine di tutti i percorsi e il codice logico.

Esempio

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

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

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

app.listen(3000);
```

Aggiunta di middleware

Le funzioni middleware sono funzioni che hanno accesso all'oggetto richiesta (req), all'oggetto risposta (res) e alla funzione middleware successiva nel ciclo richiesta-risposta dell'applicazione.

Le funzioni middleware possono eseguire qualsiasi codice, apportare modifiche agli oggetti `res` e `req` , al ciclo di risposta finale e chiamare il prossimo middleware.

Esempio molto comune di middleware è il modulo `cors` . Per aggiungere il supporto CORS, è

sufficiente installarlo, richiederlo e inserire questa riga:

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

prima di qualsiasi router o funzione di routing.

Ciao mondo

Qui creiamo un semplice server mondiale Hello usando Express. Itinerari:

- '/'
- '/ Wiki'

E per il riposo darà "404", cioè la pagina non trovata.

```
'use strict';

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

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

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

Nota: abbiamo inserito la rotta 404 come ultima rotta mentre Express impila i percorsi in ordine e li elabora in sequenza per ogni richiesta.

Leggi App Web con Express online: https://riptutorial.com/it/node-js/topic/483/app-web-con-express

Capitolo 6: Arresto grazioso

Examples

Graceful Shutdown - SIGTERM

Usando **server.close ()** e **process.exit ()** , possiamo catturare l'eccezione del server e fare un arresto regolare.

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

var server = http.createServer(function (req, res) {
  setTimeout(function () { //simulate a long request
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
  }, 4000);
}).listen(9090, function (err) {
  console.log('listening http://localhost:9090/');
  console.log('pid is ' + process.pid);
});

process.on('SIGTERM', function () {
  server.close(function () {
    process.exit(0);
  });
});
```

Leggi Arresto grazioso online: https://riptutorial.com/it/node-js/topic/5996/arresto-grazioso

Capitolo 7: Async / Await

introduzione

Async / await è un insieme di parole chiave che consente la scrittura di codice asincrono in modo procedurale senza dover fare affidamento su callback (*callback hell*) o promettere-concatenare (`.then().then().then()`).

Funziona usando la parola chiave `await` per sospendere lo stato di una funzione asincrona, fino alla risoluzione di una promessa e usando la parola chiave `async` per dichiarare tali funzioni asincrone, che restituiscono una promessa.

Async / await è disponibile da node.js 8 per impostazione predefinita o 7 utilizzando il flag `--harmony-async-await` .

Examples

Funzioni asincrone con gestione degli errori Try-Catch

Una delle migliori caratteristiche della sintassi asincrona / attendi è che lo stile di codifica try-catch standard è possibile, proprio come se si stesse scrivendo il codice sincrono.

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

Ecco un esempio con Express e promise-mysql:

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

  try {

    const connection = await pool.createConnection();

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

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

    } finally {
```

```
      pool.releaseConnection(connection);
    }

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

Confronto tra Promises e Async / Await

Funzione che usa le promesse:

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

Quindi ecco quando Async / Await entrano in azione per rendere più pulita la nostra funzione:

```
async function myAsyncFunction() {
  let result;

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

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

Quindi la parola chiave `async` sarebbe simile a write `return new Promise((resolve, reject) => {...}` .

E `await` simile a ottenere il tuo risultato in `then` richiamata.

Qui lascio una breve gif che non lascerà alcun dubbio in mente dopo averla vista:

GIF

Progressione da Callbacks

All'inizio c'erano i callback e le callback erano ok:

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

const getAirPollution = (callback) => {
```

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

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

Ma ci sono stati alcuni problemi davvero frustranti con i callback, quindi abbiamo iniziato a utilizzare le promesse.

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

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

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

Questo è stato un po 'meglio. Alla fine, abbiamo trovato async / await. Che usa ancora promesse sotto il cofano.

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

Interrompe l'esecuzione in attesa

Se la promessa non restituisce nulla, l'attività asincrona può essere completata utilizzando `await` .

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

```
}
```

Leggi Async / Await online: https://riptutorial.com/it/node-js/topic/6729/async---await

Capitolo 8: async.js

Sintassi

- **Ogni callback deve essere scritto con questa sintassi:**
- funzione callback (err, risultato [, arg1 [, ...]])
- **In questo modo, sei costretto a restituire prima l'errore, e non puoi ignorarne la gestione in seguito. `null` è la convenzione in assenza di errori**
- callback (null, myResult);
- **Le tue callback possono contenere più argomenti di *err* e *result* , ma è utile solo per un set specifico di funzioni (cascata, seq, ...)**
- callback (null, myResult, myCustomArgument);
- **E, naturalmente, inviare errori. È necessario farlo e gestire gli errori (o almeno registrarli).**
- callback (err);

Examples

Parallelo: multi-tasking

async.parallel (tasks, afterTasksCallback) eseguirà una serie di attività in parallelo e **attenderà la fine di tutte le attività** (segnalate dalla chiamata della funzione di **callback**).

Al termine delle attività, *async* chiama il callback principale con tutti gli errori e tutti i risultati delle attività.

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

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

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

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

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

Risultato: `["resultOfShortTime", "resultOfMediumTime", "resultOfLongTime"]` .

Chiama `async.parallel()` con un oggetto

È possibile sostituire il parametro dell'array *tasks* con un oggetto. In questo caso, i risultati saranno anche un oggetto **con le stesse chiavi delle attività** .

È molto utile calcolare alcune attività e trovare facilmente ogni risultato.

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

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

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

Risoluzione di più valori

Ogni funzione parallela è passata a un callback. Questo callback può restituire un errore come primo argomento o valore successivo. Se un callback viene passato a diversi valori di successo, questi risultati vengono restituiti come array.

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

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

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

Risultato:

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

.

Serie: mono-tasking indipendente

async.series (tasks, afterTasksCallback) eseguirà una serie di attività. Ogni attività viene eseguita **dopo l'altra** . **Se un'attività non riesce, *async* interrompe immediatamente l'esecuzione e passa al callback principale** .

Quando le attività sono terminate con successo, *async* chiama il callback "master" con tutti gli errori e tutti i risultati delle attività.

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

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

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

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

```
    }

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

Risultato: `["resultOfMediumTime", "resultOfShortTime", "resultOfLongTime"]` .

Chiama `async.series()` con un oggetto

È possibile sostituire il parametro dell'array *tasks* con un oggetto. In questo caso, i risultati saranno anche un oggetto **con le stesse chiavi delle attività** .

È molto utile calcolare alcune attività e trovare facilmente ogni risultato.

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

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

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

Cascata: mono-tasking dipendente

async.waterfall (tasks, afterTasksCallback) eseguirà una serie di attività. Ogni attività viene eseguita **dopo l'altra e il risultato di un'attività viene passato all'attività successiva** . As *async.series ()* , se un'attività non riesce, *async* interrompe l'esecuzione e chiama immediatamente il callback principale.

Quando le attività sono terminate con successo, *async* chiama il callback "master" con tutti gli errori e tutti i risultati delle attività.

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

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

function getUserFriendsRequest(user, callback) {
  // Another request simulate with a timeout
```

```
  setTimeout(function() {
    var friendsResult = [];

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

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

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

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

Risultato: i `results` contengono il secondo parametro di callback dell'ultima funzione della cascata, che in questo caso è `friendsResult` .

async.times (Per gestire il ciclo in modo migliore)

Per eseguire una funzione all'interno di un ciclo di node.js, è bene utilizzare una `for` ciclo per brevi cicli. Ma il ciclo è lungo, l'uso `for` ciclo `for` aumenta il tempo di elaborazione che potrebbe causare il blocco del processo del nodo. In tali scenari, è possibile utilizzare: **asycn.times**

```
function recursiveAction(n, callback)
{
    //do whatever want to do repeatedly
    callback(err, result);
}
async.times(5, function(n, next) {
    recursiveAction(n, function(err, result) {
        next(err, result);
    });
}, function(err, results) {
    // we should now have 5 result
});
```

Questo è chiamato in parallelo. Quando vogliamo chiamarlo uno alla volta, utilizzare: **async.timesSeries**

async.each (Per gestire in modo efficiente l'array di dati)

Quando vogliamo gestire una matrice di dati, è meglio usare **async.each** . Quando vogliamo

eseguire qualcosa con tutti i dati e vogliamo ottenere il callback finale una volta che tutto è stato fatto, allora questo metodo sarà utile. Questo è gestito in modo parallelo.

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

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

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

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

Per eseguire uno alla volta è possibile utilizzare **async.eachSeries**

async.series (per gestire gli eventi uno per uno)

/ In async.series, tutte le funzioni vengono eseguite in serie e gli output consolidati di ciascuna funzione vengono passati al callback finale. ad esempio /

var async = require ('async'); async.series ([function (callback) {console.log ('First Execute ..'); callback (null, 'userPersonalData');}, function (callback) {console.log ('Second Execute ..'); callback (null, 'userDependentData');}], function (err, result) {console.log (risultato);});

//Produzione:

First Execute .. Second Execute .. ['userPersonalData', 'userDependentData'] // result

Leggi async.js online: https://riptutorial.com/it/node-js/topic/3972/async-js

Capitolo 9: Autenticazione di Windows sotto node.js

Osservazioni

Esistono diversi altri APIS di Active Directory, ad esempio `activedirectory2` e `adldap` .

Examples

Utilizzando activedirectory

L'esempio qui sotto è tratto dai documenti completi, disponibili qui (GitHub) o qui (NPM) .

Installazione

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

uso

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

Leggi Autenticazione di Windows sotto node.js online: https://riptutorial.com/it/node-js/topic/10612/autenticazione-di-windows-sotto-node-js

Capitolo 10: Autoreload su modifiche

Examples

Il caricamento automatico sul codice sorgente cambia usando nodemon

Il pacchetto nodemon consente di ricaricare automaticamente il programma quando si modifica qualsiasi file nel codice sorgente.

Installazione di nodemon a livello globale

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

Installare nodemon localmente

Nel caso in cui non si desideri installarlo a livello globale

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

Utilizzando nodemon

Esegui il tuo programma con `nodemon entry.js` (o `nodemon entry`)

Questo sostituisce il solito uso del `node entry.js` (o `node entry`).

È anche possibile aggiungere l'avvio del nodemon come uno script npm, che potrebbe essere utile se si desidera fornire i parametri e non digitarli ogni volta.

Aggiungi **package.json:**

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

In questo modo puoi semplicemente utilizzare `npm start` dalla tua console.

Browsersync

Panoramica

Browsersync è uno strumento che consente la visualizzazione di file live e il ricaricamento del browser. È disponibile come pacchetto NPM .

Installazione

Per installare Browsersync devi prima installare Node.js e NPM. Per ulteriori informazioni, consultare la documentazione SO su Installazione ed esecuzione di Node.js.

Una volta impostato il tuo progetto, puoi installare Browsersync con il seguente comando:

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

Questo installerà Browsersync nella directory `node_modules` locale e lo salverà nelle dipendenze dello sviluppatore.

Se preferisci installarlo globalmente usa il flag `-g` al posto del flag `-D` .

Utenti Windows

In caso di problemi nell'installazione di Browsersync su Windows, potrebbe essere necessario installare Visual Studio in modo da poter accedere agli strumenti di compilazione per installare Browsersync. Dovrai quindi specificare la versione di Visual Studio che stai utilizzando in questo modo:

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

Questo comando specifica la versione 2013 di Visual Studio.

Uso di base

Per ricaricare automaticamente il tuo sito ogni volta che cambi un file JavaScript nel tuo progetto usa il seguente comando:

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

Sostituisci `myproject.dev` con l'indirizzo web che stai utilizzando per accedere al tuo progetto. Browsersync emetterà un indirizzo alternativo che può essere utilizzato per accedere al tuo sito tramite il proxy.

Uso avanzato

Oltre all'interfaccia della riga di comando descritta sopra, Browsersync può essere utilizzato anche con Grunt.js e Gulp.js.

Grunt.js

L'utilizzo con Grunt.js richiede un plug-in che può essere installato in questo modo:

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

Quindi aggiungerai questa linea al tuo `gruntfile.js` :

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

Gulp.js

Browsersync funziona come un modulo CommonJS , quindi non c'è bisogno di un plugin Gulp.js. Basta richiedere il modulo in questo modo:

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

Ora puoi utilizzare l' API Browsersync per configurarlo in base alle tue esigenze.

API

L'API Browsersync può essere trovata qui: https://browsersync.io/docs/api

Leggi Autoreload su modifiche online: https://riptutorial.com/it/node-js/topic/1743/autoreload-su-modifiche

Capitolo 11: Biblioteca Mongoose

Examples

Connetti a MongoDB usando Mongoose

Innanzitutto, installa Mongoose con:

```
npm install mongoose
```

Quindi, aggiungilo a `server.js` come dipendenze:

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

Successivamente, crea lo schema del database e il nome della raccolta:

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

Crea un modello e connettiti al database:

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

Quindi, avviare MongoDB ed eseguire `server.js` utilizzando `node server.js`

Per verificare se ci siamo connessi con successo al database, possiamo usare gli eventi `open` , `error` dall'oggetto `mongoose.connection` .

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

Salva i dati su MongoDB usando Mongoose e Express.js Routes

Impostare

Innanzitutto, installa i pacchetti necessari con:

```
npm install express cors mongoose
```

Codice

Quindi, aggiungi dipendenze al file `server.js` , crea lo schema del database e il nome della raccolta, crea un server Express.js e connettiti a MongoDB:

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

var app = express();

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

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

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

Ora aggiungi i percorsi Express.js che useremo per scrivere i dati:

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

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

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

Qui la variabile di `query` sarà il parametro `<query>` della richiesta HTTP in entrata, che verrà salvata in MongoDB:

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

Se si verifica un errore durante il tentativo di scrittura su MongoDB, riceverai un messaggio di errore sulla console. Se tutto è andato a buon fine, vedrai i dati salvati in formato JSON sulla pagina.

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

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

Ora, è necessario avviare MongoDB ed eseguire il file `server.js` utilizzando `node server.js` .

uso

Per utilizzare questo per salvare i dati, vai al seguente URL nel browser:

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

Dove `<query>` è la nuova richiesta che desideri salvare.

Esempio:

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

Uscita in formato JSON:

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

Trova i dati in MongoDB usando i percorsi Mongoose e Express.js

Impostare

Innanzitutto, installa i pacchetti necessari con:

```
npm install express cors mongoose
```

Codice

Quindi, aggiungi dipendenze a `server.js` , crea lo schema del database e il nome della raccolta, crea un server Express.js e connettiti a MongoDB:

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

var app = express();

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

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

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

Ora aggiungi le rotte Express.js che useremo per interrogare i dati:

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

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

Supponiamo che i seguenti documenti siano presenti nella collezione nel modello:

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

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

E l'obiettivo è trovare e visualizzare tutti i documenti contenenti `"JavaScript is Awesome"` sotto la chiave `"request"` .

Per fare ciò, avviare MongoDB ed eseguire `server.js` con `node server.js` :

uso

Per utilizzare questo per trovare i dati, vai al seguente URL in un browser:

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

Dove `<query>` è la query di ricerca.

Esempio:

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

Produzione:

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

Trova i dati in MongoDB usando Mongoose, Express.js Routes e $ text Operator

Impostare

Innanzitutto, installa i pacchetti necessari con:

```
npm install express cors mongoose
```

Codice

Quindi, aggiungi dipendenze a `server.js` , crea lo schema del database e il nome della raccolta, crea un server Express.js e connettiti a MongoDB:

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

var app = express();

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

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

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

Ora aggiungi le rotte Express.js che useremo per interrogare i dati:

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

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

Supponiamo che i seguenti documenti siano presenti nella collezione nel modello:

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

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

E che l'obiettivo è trovare e visualizzare tutti i documenti contenenti solo la parola `"JavaScript"` sotto la chiave `"request"` .

Per fare ciò, prima crea un *indice di testo* per `"request"` nella raccolta. Per questo, aggiungi il seguente codice a `server.js` :

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

E sostituire:

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

Con:

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

Qui, stiamo usando `$text` e `$search` operatori MongoDB per trovare tutti i documenti nella collection `collectionName` che contiene almeno una parola dalla query di ricerca specificata.

uso

Per utilizzare questo per trovare i dati, vai al seguente URL in un browser:

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

Dove `<query>` è la query di ricerca.

Esempio:

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

Produzione:

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

Indici nei modelli.

MongoDB supporta indici secondari. In Mongoose, definiamo questi indici all'interno del nostro schema. La definizione di indici a livello di schema è necessaria quando è necessario creare indici composti.

Connessione Mongoose

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

Creazione di uno schema di base

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

var usersModel = db.model('users', usersSchema);
module.exports = usersModel;
```

Per impostazione predefinita, mangusta aggiunge due nuovi campi nel nostro modello, anche quando quelli non sono definiti nel modello. Questi campi sono:

_ID

Mongoose assegna a ciascuno degli schemi un campo _id per impostazione predefinita se uno non viene passato al costruttore Schema. Il tipo assegnato è un ObjectId che coincide con il comportamento predefinito di MongoDB. Se non vuoi che un _id venga aggiunto al tuo schema, puoi disabilitarlo usando questa opzione.

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

__v o versionKey

VersionKey è una proprietà impostata su ciascun documento quando viene creata da Mongoose. Questo valore di chiavi contiene la revisione interna del documento. Il nome di questa proprietà del documento è configurabile.

Puoi disabilitare facilmente questo campo nella configurazione del modello:

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

Indici composti

Possiamo creare altri indici oltre a quelli creati da Mangusta.

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

In questo caso il nostro modello ha altri due indici, uno per il campo username e l'altro per il campo email. Ma possiamo creare indici composti.

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

Impatto sulle prestazioni dell'indice

Per impostazione predefinita, la mangusta chiama sempre il parametro sureIndex per ciascun

indice in sequenza ed emette un evento 'index' sul modello quando tutte le chiamate sureIndex hanno avuto esito positivo o quando si è verificato un errore.

In MongoDB, sureIndex è deprecato dalla versione 3.0.0, ora è un alias per createIndex.

Si consiglia di disabilitare il comportamento impostando l'opzione autoIndex dello schema su false o globalmente sulla connessione impostando l'opzione config.autoIndex su false.

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

Funzioni utili di Mongoose

Mongoose contiene alcune funzioni incorporate che si basano sul `find()` standard `find()` .

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

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

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

trova i dati in mongodb usando le promesse

Impostare

Innanzitutto, installa i pacchetti necessari con:

```
npm install express cors mongoose
```

Codice

Quindi, aggiungi dipendenze a `server.js` , crea lo schema del database e il nome della raccolta, crea un server Express.js e connettiti a MongoDB:

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

var app = express();

var schemaName = new Schema({
    request: String,
    time: Number
```

```
}, {
    collection: 'collectionName'
});

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

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

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

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

Ora aggiungi le rotte Express.js che useremo per interrogare i dati:

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

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

Supponiamo che i seguenti documenti siano presenti nella collezione nel modello:

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

E l'obiettivo è trovare e visualizzare tutti i documenti contenenti `"JavaScript is Awesome"` sotto la chiave `"request"` .

Per fare ciò, avviare MongoDB ed eseguire `server.js` con `node server.js` :

uso

Per utilizzare questo per trovare i dati, vai al seguente URL in un browser:

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

Dove `<query>` è la query di ricerca.

Esempio:

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

Produzione:

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

Leggi Biblioteca Mongoose online: https://riptutorial.com/it/node-js/topic/3486/biblioteca-mongoose

Capitolo 12: Buon stile di codifica

Osservazioni

Consiglierei ad un principiante di iniziare con questo stile di codifica. E se qualcuno può suggerire un modo migliore (ps ho optato per questa tecnica e sta funzionando in modo efficiente per me in un'app utilizzata da più di 100k utenti), sentitevi liberi per qualsiasi suggerimento. TIA.

Examples

Programma base per la registrazione

Attraverso questo esempio, verrà spiegato di dividere il codice **node.js** in diversi **moduli / cartelle** per una migliore sottotollerabilità. Seguendo questa tecnica è più facile per gli altri sviluppatori capire il codice dato che può riferirsi direttamente al file interessato invece di passare attraverso l'intero codice. L'uso principale è quando si lavora in una squadra e un nuovo sviluppatore si unisce in una fase successiva, per lui sarà più facile assimilare il codice stesso.

index.js : - Questo file gestirà la connessione al server.

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

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

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

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

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

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

config.js : -Questo file gestirà tutti i **param** relativi alla configurazione che rimarranno gli stessi in tutto.

```
var config = {
VERSION: 1,
BUILD: 1,
URL: 'http://127.0.0.1',
```

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

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

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

module.exports = config;
```

user.js : - File di modello in cui è definito lo schema

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

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

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

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

userController : - Questo file contiene la funzione per l'utente registrati

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

//Controller for User
var UserController = {

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

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

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

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

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

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

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

```
                            }

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

}

module.exports = UserController;
```

userRoutes.js : - Questo è il percorso per userController

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

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

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

return router;

}

module.exports = UserRoutes;
```

L'esempio sopra può sembrare troppo grande ma se un principiante su node.js con una piccola miscela di conoscenza espressa cerca di farlo, troverà la cosa facile e veramente utile.

Leggi Buon stile di codifica online: https://riptutorial.com/it/node-js/topic/6489/buon-stile-di-codifica

Capitolo 13: CLI

Sintassi

- nodo [opzioni] [opzioni v8] [script.js | -e "script"] [argomenti]

Examples

Opzioni della riga di comando

```
-v, --version
```

Aggiunto in: v0.1.3 Stampa della versione del nodo.

```
-h, --help
```

Aggiunto in: v0.1.3 Opzioni della riga di comando del nodo di stampa. L'output di questa opzione è meno dettagliato di questo documento.

```
-e, --eval "script"
```

Aggiunto in: v0.5.2 Valuta il seguente argomento come JavaScript. I moduli che sono predefiniti in REPL possono essere utilizzati anche in script.

```
-p, --print "script"
```

Aggiunto in: v0.6.4 Identico a -e ma stampa il risultato.

```
-c, --check
```

Aggiunto in: v5.0.0 Sintassi controlla lo script senza eseguire.

```
-i, --interactive
```

Aggiunto in: v0.7.7 Apre REPL anche se stdin non sembra essere un terminale.

```
-r, --require module
```

Aggiunto in: v1.6.0 Precarica il modulo specificato all'avvio.

Segue le regole di risoluzione dei moduli di require (). il modulo può essere un percorso per un file o un nome di modulo nodo.

```
--no-deprecation
```

Aggiunto in: v0.8.0 avvisi di deprecazione del silenzio.

```
--trace-deprecation
```

Aggiunto in: v0.8.0 Stampa tracce di stack per le deprecazioni.

```
--throw-deprecation
```

Aggiunto in: v0.11.14 Lancia gli errori per le deprecazioni.

```
--no-warnings
```

Aggiunto in: v6.0.0 Silenzia tutti gli avvisi di processo (incluse le deprecazioni).

```
--trace-warnings
```

Aggiunto in: v6.0.0 Stampa tracce di stack per gli avvisi di processo (incluse le deprecazioni).

```
--trace-sync-io
```

Aggiunto in: v2.1.0 Stampa una traccia di stack ogni volta che viene rilevato I / O sincrono dopo il primo turno del ciclo degli eventi.

```
--zero-fill-buffers
```

Aggiunto in: v6.0.0 Automaticamente zero riempie tutte le istanze di Buffer e SlowBuffer appena allocate.

```
--preserve-symlinks
```

Aggiunto in: v6.3.0 Indica al modulo loader di conservare i collegamenti simbolici durante la risoluzione e la memorizzazione nella cache dei moduli.

Per impostazione predefinita, quando Node.js carica un modulo da un percorso che è collegato simbolicamente a una diversa posizione su disco, Node.js cancellerà il collegamento e utilizzerà l'effettivo "percorso reale" su disco del modulo come identificativo e come percorso root per individuare altri moduli di dipendenza. Nella maggior parte dei casi, questo comportamento predefinito è accettabile. Tuttavia, quando si utilizzano dipendenze peer collegate simbolicamente, come illustrato nell'esempio seguente, il comportamento predefinito causa l'emissione di un'eccezione se il moduloA tenta di richiedere moduleB come dipendenza peer:

```
{appDir}
 ├── app
 │   ├── index.js
 │   └── node_modules
 │       ├── moduleA -> {appDir}/moduleA
 │       └── moduleB
 │           ├── index.js
```

```
│           └── package.json
└── moduleA
    ├── index.js
    └── package.json
```

L'indicatore della riga di comando --preserve-symlinks ordina a Node.js di utilizzare il percorso del collegamento simbolico per i moduli anziché il percorso reale, consentendo di trovare le dipendenze peer collegate simbolicamente.

Si noti, tuttavia, che l'utilizzo di --preserve-symlinks può avere altri effetti collaterali. In particolare, i moduli nativi collegati simbolicamente non riescono a caricare se sono collegati da più di una posizione nell'albero delle dipendenze (Node.js li vedrebbe come due moduli separati e tenterebbe di caricare il modulo più volte, causando un'eccezione).

```
--track-heap-objects
```

Aggiunto in: v2.4.0 Traccia le allocazioni degli oggetti heap per le istantanee heap.

```
--prof-process
```

Aggiunto in: v6.0.0 Output del processo v8 del processo generato utilizzando l'opzione v8 --prof.

```
--v8-options
```

Aggiunto in: v0.1.3 Stampa le opzioni della riga di comando v8.

Nota: le opzioni v8 consentono alle parole di essere separate da entrambi i trattini (-) o caratteri di sottolineatura (_).

Ad esempio, --stack-trace-limit è equivalente a --stack_trace_limit.

```
--tls-cipher-list=list
```

Aggiunto in: v4.0.0 Specificare un elenco di crittografia TLS predefinito alternativo. (Richiede Node.js per essere costruito con supporto crittografico. (Predefinito))

```
--enable-fips
```

Aggiunto in: v6.0.0 Abilita crypto compatibile con FIPS all'avvio. (Richiede che Node.js sia compilato con ./configure --openssl-fips)

```
--force-fips
```

Aggiunto in: v6.0.0 Forza la crittografia compatibile con FIPS all'avvio. (Non può essere disabilitato dal codice di script.) (Stessi requisiti di --enable-fips)

```
--icu-data-dir=file
```

Aggiunto in: v0.11.15 Specificare il percorso di caricamento dei dati ICU. (sostituisce NODE_ICU_DATA)

```
Environment Variables

NODE_DEBUG=module[,…]
```

Aggiunto in: v0.1.32 ',' - elenco separato dei moduli principali che dovrebbero stampare le informazioni di debug.

```
NODE_PATH=path[:…]
```

Aggiunto in: v0.1.32 ':' - elenco separato di directory con prefisso al percorso di ricerca del modulo.

Nota: su Windows, questa è una ';' - lista separata invece.

```
NODE_DISABLE_COLORS=1
```

Aggiunto in: v0.3.0 Se impostato su 1 colori non verrà utilizzato in REPL.

```
NODE_ICU_DATA=file
```

Aggiunto in: v0.11.15 Percorso dati per i dati ICU (oggetto Intl). Estenderà i dati collegati quando compilati con supporto small-icu.

```
NODE_REPL_HISTORY=file
```

Aggiunto in: v5.0.0 Percorso del file utilizzato per memorizzare la cronologia REPL persistente. Il percorso predefinito è ~ / .node_repl_history, che viene sovrascritto da questa variabile. L'impostazione del valore su una stringa vuota ("" o "") disabilita la cronologia REPL persistente.

Leggi CLI online: https://riptutorial.com/it/node-js/topic/6013/cli

Capitolo 14: Codice Node.js per STDIN e STDOUT senza utilizzare alcuna libreria

introduzione

Questo è un semplice programma in node.js a cui prende input dall'utente e lo stampa sulla console.

L'oggetto del **processo** è un globale che fornisce informazioni e controlla il processo Node.js corrente. Come globale, è sempre disponibile per le applicazioni Node.js senza usare require ().

Examples

Programma

La proprietà **process.stdin** restituisce un flusso leggibile equivalente o associato a stdin.

La proprietà **process.stdout** restituisce un flusso Writable equivalente o associato a stdout.

```
process.stdin.resume()
console.log('Enter the data to be displayed ');
process.stdin.on('data', function(data) { process.stdout.write(data) })
```

Leggi Codice Node.js per STDIN e STDOUT senza utilizzare alcuna libreria online: https://riptutorial.com/it/node-js/topic/8961/codice-node-js-per-stdin-e-stdout-senza-utilizzare-alcuna-libreria

Capitolo 15: Come vengono caricati i moduli

Examples

Modalità globale

Se hai installato il nodo usando la directory predefinita, mentre in modalità globale, NPM installa i pacchetti in `/usr/local/lib/node_modules` . Se nella shell si digita quanto segue, NPM cercherà, scaricherà e installerà l'ultima versione del pacchetto chiamato sax all'interno della directory `/usr/local/lib/node_modules/express` :

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

Assicurarsi di disporre di diritti di accesso sufficienti per la cartella. Questi moduli saranno disponibili per tutto il processo del nodo che verrà eseguito su quella macchina

Nell'installazione in modalità locale. Npm scaricherà e installerà i moduli nelle attuali cartelle di lavoro creando una nuova cartella denominata `node_modules` ad esempio se ti trovi in `/home/user/apps/my_app` verrà creata una nuova cartella chiamata `node_modules` `/home/user/apps/my_app/node_modules` se non è già esistente

Caricamento dei moduli

Quando si fa riferimento al modulo nel codice, il nodo ricerca prima la cartella `node_module` all'interno della cartella di riferimento `node_module` richiesta Se il nome del modulo non è relativo e non è un modulo principale, il nodo cercherà di trovarlo all'interno della cartella `node_modules` nell'attuale directory. Ad esempio, se fai quanto segue, Node proverà a cercare il file `./node_modules/myModule.js` :

```
var myModule = require('myModule.js');
```

Se il nodo non riesce a trovare il file, cercherà all'interno della cartella genitore chiamata `../node_modules/myModule.js` . Se fallisce di nuovo, proverà la cartella genitore e continuerà a discendere fino a raggiungere la radice o trovare il modulo richiesto.

Se lo desideri, puoi anche omettere l'estensione `.js` , nel qual caso il nodo aggiungerà l'estensione `.js` e cercherà il file.

Caricamento di un modulo di cartelle

Puoi usare il percorso di una cartella per caricare un modulo come questo:

```
var myModule = require('./myModuleDir');
```

In tal caso, il nodo cercherà all'interno di quella cartella. Il nodo presumerà che questa cartella sia un pacchetto e proverà a cercare una definizione del pacchetto. Quella definizione del pacchetto dovrebbe essere un file denominato `package.json` . Se quella cartella non contiene un file di definizione del pacchetto denominato `package.json` , il punto di ingresso del pacchetto assumerà il valore predefinito di `index.js` , e il Node guarderà, in questo caso, per un file sotto il percorso `./myModuleDir/index.js` .

L'ultima risorsa, se il modulo non si trova in nessuna delle cartelle, è la cartella di installazione del modulo globale.

Leggi Come vengono caricati i moduli online: https://riptutorial.com/it/node-js/topic/7738/come-vengono-caricati-i-moduli

Capitolo 16: Comunicazione Arduino con nodeJs

introduzione

Modo per mostrare come Node.Js può comunicare con Arduino Uno.

Examples

Comunicazione Node Js con Arduino via serialport

Codice js del nodo

Esempio per iniziare questo argomento è il server Node.js che comunica con Arduino tramite serialport.

```
npm install express --save
npm install serialport --save
```

Esempio app.js:

```
const express = require('express');
const app = express();
var SerialPort = require("serialport");

var port = 3000;

var arduinoCOMPort = "COM3";

var arduinoSerialPort = new SerialPort(arduinoCOMPort, {
 baudrate: 9600
});

arduinoSerialPort.on('open',function() {
  console.log('Serial Port ' + arduinoCOMPort + ' is opened.');
});

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

    return res.send('Working');

})

app.get('/:action', function (req, res) {

   var action = req.params.action || req.param('action');

    if(action == 'led'){
        arduinoSerialPort.write("w");
        return res.send('Led light is on!');
```

```
    }
    if(action == 'off') {
        arduinoSerialPort.write("t");
        return res.send("Led light is off!");
    }

    return res.send('Action: ' + action);

});

app.listen(port, function () {
  console.log('Example app listening on port http://0.0.0.0:' + port + '!');
});
```

Avvio del server Express di esempio:

```
node app.js
```

Codice Arduino

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.

  Serial.begin(9600); // Begen listening on port 9600 for serial

  pinMode(LED_BUILTIN, OUTPUT);

  digitalWrite(LED_BUILTIN, LOW);
}

// the loop function runs over and over again forever
void loop() {

   if(Serial.available() > 0) // Read from serial port
    {
      char ReaderFromNode; // Store current character
      ReaderFromNode = (char) Serial.read();
      convertToState(ReaderFromNode); // Convert character to state
    }
  delay(1000);
}

void convertToState(char chr) {
  if(chr=='o'){
    digitalWrite(LED_BUILTIN, HIGH);
    delay(100);
  }
  if(chr=='f'){
    digitalWrite(LED_BUILTIN, LOW);
    delay(100);
  }
}
```

Cominciando

1. Collega l'arduino alla tua macchina.
2. Avvia il server

Controlla la compilazione guidata tramite il nodo js express server.

Per accendere il led:

```
http://0.0.0.0:3000/led
```

Per spegnere il led:

```
http://0.0.0.0:3000/off
```

Leggi Comunicazione Arduino con nodeJs online: https://riptutorial.com/it/node-js/topic/10509/comunicazione-arduino-con-nodejs

Capitolo 17: Comunicazione client-server

Examples

/ w Express, jQuery e Jade

```
//'client.jade'

//a button is placed down; similar in HTML
button(type='button', id='send_by_button') Modify data

    #modify Lorem ipsum Sender

    //loading jQuery; it can be done from an online source as well
    script(src='./js/jquery-2.2.0.min.js')

    //AJAX request using jQuery
    script
        $(function () {
            $('#send_by_button').click(function (e) {
                e.preventDefault();

                //test: the text within brackets should appear when clicking on said button
                //window.alert('You clicked on me.  - jQuery');

                //a variable and a JSON initialized in the code
                var predeclared = "Katamori";
                var data = {
                    Title: "Name_SenderTest",
                    Nick: predeclared,
                    FirstName: "Zoltan",
                    Surname: "Schmidt"
                };

                //an AJAX request with given parameters
                $.ajax({
                    type: 'POST',
                    data: JSON.stringify(data),
                    contentType: 'application/json',
                    url: 'http://localhost:7776/domaintest',

                    //on success, received data is used as 'data' function input
                    success: function (data) {
                        window.alert('Request sent; data received.');

                        var jsonstr = JSON.stringify(data);
                        var jsonobj = JSON.parse(jsonstr);

                        //if the 'nick' member of the JSON does not equal to the predeclared
string (as it was initialized), then the backend script was executed, meaning that
communication has been established
                        if(data.Nick != predeclared){
                            document.getElementById("modify").innerHTML = "JSON changed!\n" +
jsonstr;
                        };

                    }
```

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

//'domaintest_route.js'

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

//an Express router listening to GET requests - in this case, it's empty, meaning that nothing
is displayed when you reach 'localhost/domaintest'
router.get('/', function(req, res, next) {
});

//same for POST requests - notice, how the AJAX request above was defined as POST
router.post('/', function(req, res) {
    res.setHeader('Content-Type', 'application/json');

    //content generated here
    var some_json = {
        Title: "Test",
        Item: "Crate"
    };

    var result = JSON.stringify(some_json);

    //content got 'client.jade'
    var sent_data = req.body;
    sent_data.Nick = "ttony33";

    res.send(sent_data);

});


module.exports = router;
```

// basato su un gist usato personalmente: https://gist.github.com/Katamori/5c9850f02e4baf6e9896

Leggi Comunicazione client-server online: https://riptutorial.com/it/node-js/topic/6222/comunicazione-client-server

Capitolo 18: Comunicazione Socket.io

Examples

"Ciao mondo!" con i messaggi socket

Installa i moduli del nodo

```
npm install express
npm install socket.io
```

Node.js server

```
const express = require('express');
const app = express();
const server = app.listen(3000,console.log("Socket.io Hello World server started!"));
const io = require('socket.io')(server);

io.on('connection', (socket) => {
    //console.log("Client connected!");
    socket.on('message-from-client-to-server', (msg) => {
        console.log(msg);
    })
    socket.emit('message-from-server-to-client', 'Hello World!');
});
```

Client browser

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Hello World with Socket.io</title>
  </head>
  <body>
    <script src="https://cdn.socket.io/socket.io-1.4.5.js"></script>
    <script>
      var socket = io("http://localhost:3000");
      socket.on("message-from-server-to-client", function(msg) {
          document.getElementById('message').innerHTML = msg;
      });
      socket.emit('message-from-client-to-server', 'Hello World!');
    </script>
    <p>Socket.io Hello World client started!</p>
    <p id="message"></p>
  </body>
</html>
```

Leggi Comunicazione Socket.io online: https://riptutorial.com/it/node-js/topic/4261/comunicazione-socket-io

Capitolo 19: Connetti a Mongodb

introduzione

MongoDB è un programma di database orientato ai documenti multipiattaforma gratuito e open source. Classificato come programma di database NoSQL, MongoDB utilizza documenti simili a JSON con schemi.

Per maggiori dettagli vai su https://www.mongodb.com/

Sintassi

- MongoClient.connect ('mongodb: //127.0.0.1: 27017 / crud', function (err, db) {// fa womething qui});

Examples

Semplice esempio per collegare mongoDB da Node.JS

```
MongoClient.connect('mongodb://localhost:27017/myNewDB',function (err,db) {
    if(err)
        console.log("Unable to connect DB. Error: " + err)
    else
        console.log('Connected to DB');

    db.close();
});
```

myNewDB è un nome DB, se non esiste nel database, verrà creato automaticamente con questa chiamata.

Un modo semplice per collegare mongoDB con il core Node.JS

```
  var MongoClient = require('mongodb').MongoClient;

    //connection with mongoDB
    MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {
          //check the connection
 if(err){
           console.log("connection failed.");
}else{
                    console.log("successfully connected to mongoDB.");
    });
```

Leggi Connetti a Mongodb online: https://riptutorial.com/it/node-js/topic/6280/connetti-a-mongodb

Capitolo 20: Consegna HTML o qualsiasi altro tipo di file

Sintassi

- response.sendFile (fileName, options, function (err) {});

Examples

Consegna HTML al percorso specificato

Ecco come creare un server Express e servire `index.html` per impostazione predefinita (percorso vuoto `/`) e `page1.html` per `/page1` percorso `/page1` .

Struttura delle cartelle

```
project root
|    server.js
|____views
     |    index.html
     |    page1.html
```

server.js

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

// deliver index.html if no file is requested
app.get("/", function (request, response) {
  response.sendFile(path.join(__dirname, 'views/index.html'));
});

// deliver page1.html if page1 is requested
app.get('/page1', function(request, response) {
    response.sendFile(path.join(__dirname, 'views', 'page1.html', function(error) {
        if (error) {
            // do something in case of error
            console.log(err);
            response.end(JSON.stringify({error:"page not found"}));
        }
    });
});

app.listen(8080);
```

Si noti che `sendFile()` semplicemente lo streaming di un file statico come risposta, non offrendo

alcuna possibilità di modificarlo. Se stai servendo un file HTML e vuoi includere dati dinamici con esso, dovrai utilizzare un *motore di template* come Pug, Moustache o EJS.

Leggi Consegna HTML o qualsiasi altro tipo di file online: https://riptutorial.com/it/node-js/topic/6538/consegna-html-o-qualsiasi-altro-tipo-di-file

Capitolo 21: Creazione di API con Node.js

Examples

Ottieni API usando Express

`Node.js` possono essere facilmente costruite nel framework web `Express` .

L'esempio seguente crea una semplice API `GET` per elencare tutti gli utenti.

Esempio

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

var users =[{
        id: 1,
        name: "John Doe",
        age : 23,
        email: "john@doe.com"
    }];

// GET /api/users
app.get('/api/users', function(req, res){
    return res.json(users);    //return response as JSON
});

app.listen('3000', function(){
    console.log('Server listening on port 3000');
});
```

API POST utilizzando Express

L'esempio seguente crea API `POST` usando `Express` . Questo esempio è simile all'esempio `GET` tranne l'uso di `body-parser` che analizza i dati del post e lo aggiunge a `req.body` .

Esempio

```
var express = require('express');
var app = express();
// for parsing the body in POST request
var bodyParser = require('body-parser');

var users =[{
    id: 1,
    name: "John Doe",
    age : 23,
    email: "john@doe.com"
}];

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
```

```
// GET /api/users
app.get('/api/users', function(req, res){
    return res.json(users);
});


/* POST /api/users
    {
        "user": {
           "id": 3,
            "name": "Test User",
            "age" : 20,
            "email": "test@test.com"
        }
    }
*/
app.post('/api/users', function (req, res) {
    var user = req.body.user;
    users.push(user);

    return res.send('User has been added successfully');
});

app.listen('3000', function(){
    console.log('Server listening on port 3000');
});
```

Leggi Creazione di API con Node.js online: https://riptutorial.com/it/node-js/topic/5991/creazione-di-api-con-node-js

Capitolo 22: Creazione di una libreria Node.js che supporti entrambe le promesse e le callback first-error

introduzione

A molte persone piace lavorare con promesse e / o asincroni / attendere la sintassi, ma quando si scrive un modulo sarebbe utile che alcuni programmatori supportino anche i metodi classici di stile callback. Piuttosto che creare due moduli, o due set di funzioni, o avere il programmatore che promette il tuo modulo, il tuo modulo può supportare entrambi i metodi di programmazione su uno usando asCallback () o Q's (Windows) di Q. ().

Examples

Modulo di esempio e programma corrispondente che utilizza Bluebird

math.js

```
'use strict';

const Promise = require('bluebird');

module.exports = {

  // example of a callback-only method
  callbackSum: function(a, b, callback) {
    if (typeof a !== 'number')
      return callback(new Error('"a" must be a number'));
    if (typeof b !== 'number')
      return callback(new Error('"b" must be a number'));

    return callback(null, a + b);
  },

  // example of a promise-only method
  promiseSum: function(a, b) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error('"a" must be a number'));
      if (typeof b !== 'number')
        return reject(new Error('"b" must be a number'));
      resolve(a + b);
    });
  },

  // a method that can be used as a promise or with callbacks
  sum: function(a, b, callback) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error('"a" must be a number'));
```

```
      if (typeof b !== 'number')
        return reject(new Error('"b" must be a number'));
      resolve(a + b);
    }).asCallback(callback);
  },

};
```

index.js

```
'use strict';

const math = require('./math');


// classic callbacks

math.callbackSum(1, 3, function(err, result) {
  if (err)
    console.log('Test 1: ' + err);
  else
    console.log('Test 1: the answer is ' + result);
});

math.callbackSum(1, 'd', function(err, result) {
  if (err)
    console.log('Test 2: ' + err);
  else
    console.log('Test 2: the answer is ' + result);
});


// promises

math.promiseSum(2, 5)
.then(function(result) {
  console.log('Test 3: the answer is ' + result);
})
.catch(function(err) {
  console.log('Test 3: ' + err);
});

math.promiseSum(1)
.then(function(result) {
  console.log('Test 4: the answer is ' + result);
})
.catch(function(err) {
  console.log('Test 4: ' + err);
});


// promise/callback method used like a promise

math.sum(8, 2)
.then(function(result) {
  console.log('Test 5: the answer is ' + result);
})
.catch(function(err) {
  console.log('Test 5: ' + err);
});
```

```
// promise/callback method used with callbacks

math.sum(7, 11, function(err, result) {
  if (err)
    console.log('Test 6: ' + err);
  else
    console.log('Test 6: the answer is ' + result);
});


// promise/callback method used like a promise with async/await syntax

(async () => {

  try {
    let x = await math.sum(6, 3);
    console.log('Test 7a: ' + x);

    let y = await math.sum(4, 's');
    console.log('Test 7b: ' + y);

  } catch(err) {
    console.log(err.message);
  }

})();
```

Leggi Creazione di una libreria Node.js che supporti entrambe le promesse e le callback first-error online: https://riptutorial.com/it/node-js/topic/9874/creazione-di-una-libreria-node-js-che-supporti-entrambe-le-promesse-e-le-callback-first-error

Capitolo 23: Database (MongoDB con Mongoose)

Examples

Connessione Mongoose

Assicurati di avere prima mongodb in esecuzione! `mongod --dbpath data/`

package.json

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

server.js (ECMA 6)

```
import mongoose from 'mongoose';

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

server.js (ECMA 5.1)

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

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

Modello

Definisci il tuo modello (i):

app / models / user.js (ECMA 6)

```
import mongoose from 'mongoose';

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

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

export default User;
```

app / model / user.js (ECMA 5.1)

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

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

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

module.exports = User
```

Inserisci dati

ECMA 6:

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

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

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

ECMA5.1:

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

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

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

Leggi i dati

ECMA6:

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

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

```
});
```

ECMA5.1:

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

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

Leggi Database (MongoDB con Mongoose) online: https://riptutorial.com/it/node-js/topic/6411/database--mongodb-con-mongoose-

Capitolo 24: Debug dell'applicazione Node.js

Examples

Core node.js debugger e node inspector

Utilizzo del debugger principale

Node.js fornisce una build in utilità di debug non grafica. Per avviare la compilazione nel debugger, avviare l'applicazione con questo comando:

```
node debug filename.js
```

Si consideri la seguente semplice applicazione Node.js contenuta nel `debugDemo.js`

```
'use strict';

function addTwoNumber(a, b){
// function returns the sum of the two numbers
debugger
  return a + b;
}

var result = addTwoNumber(5, 9);
console.log(result);
```

Il `debugger` parole chiave bloccherà il debugger in quel punto del codice.

Riferimento del comando

1. Stepping

```
cont, c - Continue execution
next, n - Step next
step, s - Step in
out, o - Step out
```

2. I punti di interruzione

```
setBreakpoint(), sb() - Set breakpoint on current line
setBreakpoint(line), sb(line) - Set breakpoint on specific line
```

Per eseguire il debug del codice precedente, eseguire il seguente comando

```
node debug debugDemo.js
```

Una volta eseguiti i comandi precedenti, vedrai il seguente output. Per uscire dall'interfaccia del debugger, digitare `process.exit()`

```
ankuranand:~/workspace/nodejs/nodejsDebugging $ node debug debugDemo.js
< Debugger listening on port 5858
debug> . ok
break in debugDemo.js:3
  1 // A Demo Code Showing the basic capabilities of the nodejs  debugging module
  2
> 3 'use strict';
  4
  5 function addTwoNumber(a, b){
debug> n
break in debugDemo.js:11
  9 }
 10
>11 let result = addTwoNumber(5, 9);
 12 console.log(result);
 13
debug> c
break in debugDemo.js:7
  5 function addTwoNumber(a, b){
  6 // function returns the sum of the two numbers
> 7 debugger
  8   return a + b;
  9 }
debug> c
< 14
debug> process.exit()
ankuranand:~/workspace/nodejs/nodejsDebugging $
```

Utilizzare il comando `watch(expression)` per aggiungere la variabile o l'espressione di cui si desidera visualizzare il valore e `restart` per riavviare l'app e il debug.

Usa `repl` per inserire il codice in modo interattivo. La modalità repl ha lo stesso contesto della linea di cui si sta eseguendo il debug. Ciò consente di esaminare il contenuto delle variabili e testare le linee di codice. Premi `Ctrl+C` per lasciare il debug repl.

Utilizzando l'ispettore del nodo incorporato

v6.3.0

Puoi eseguire l'ispettore integrato in v8 del nodo! Il plug-in ispettore nodo non è più necessario.

Basta passare il flag di ispezione e ti verrà fornito un URL per l'ispettore

```
node --inspect server.js
```

Utilizzando l'ispettore Node

Installa l'ispettore del nodo:

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

Esegui la tua app con il comando node-debug:

```
node-debug filename.js
```

Successivamente, premi in Chrome:

```
http://localhost:8080/debug?port=5858
```

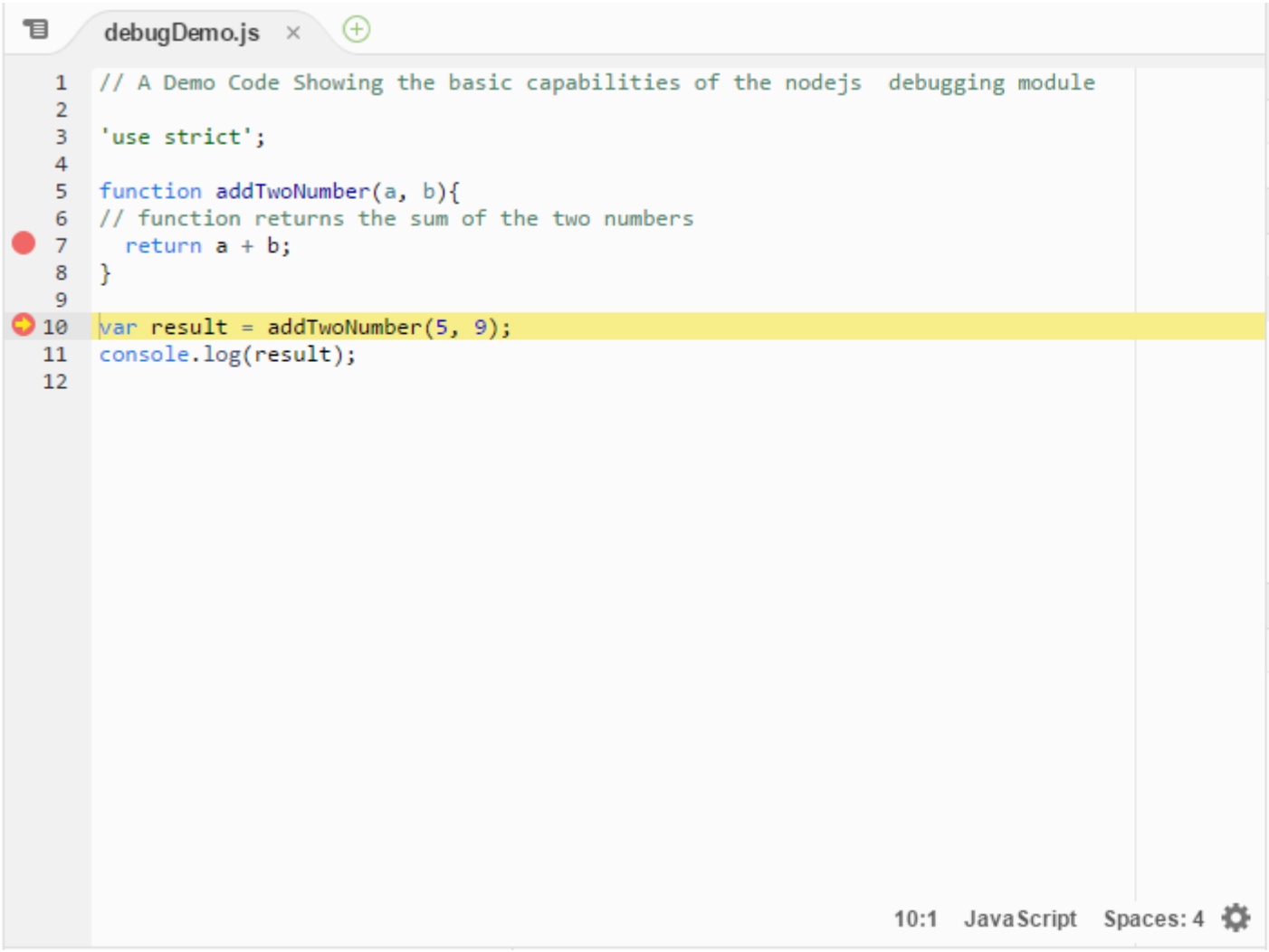
A volte la porta 8080 potrebbe non essere disponibile sul tuo computer. Potresti ricevere il seguente errore:

> Impossibile avviare il server 0.0.0.0:8080. Errore: ascolta EACCES.

In questo caso, avviare l'ispettore del nodo su una porta diversa usando il seguente comando.

```
$node-inspector --web-port=6500
```

Vedrai qualcosa di simile a questo:



Leggi Debug dell'applicazione Node.js online: https://riptutorial.com/it/node-js/topic/5900/debug-dell-applicazione-node-js

Capitolo 25: Debug remoto in Node.JS

Examples

NodeJS esegue la configurazione

Per configurare il debug remoto dei nodi, esegui semplicemente il processo del nodo con il flag `--debug` . È possibile aggiungere una porta su cui eseguire il debugger utilizzando `--debug=<port>` .

Quando il processo del tuo nodo si avvia, dovresti vedere il messaggio

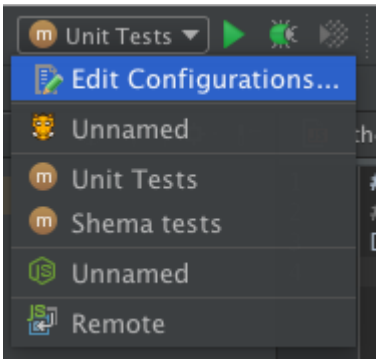
```
Debugger listening on port <port>
```

Che ti dirà che tutto è buono per andare.

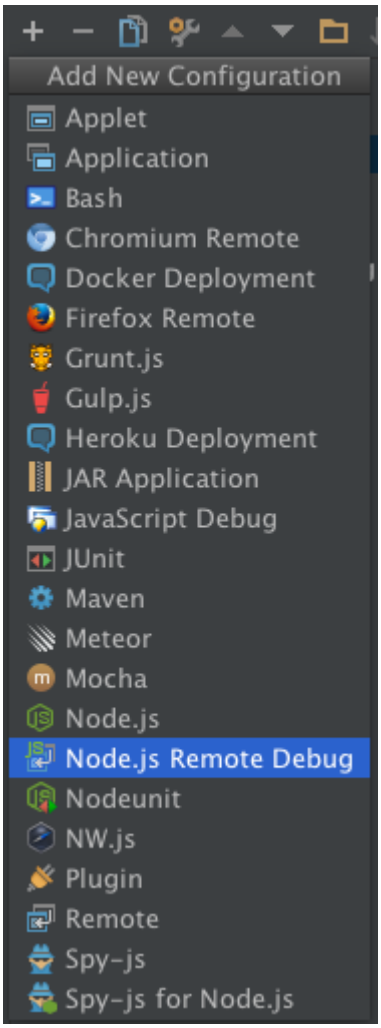
Quindi imposti il target di debug remoto nel tuo IDE specifico.

Configurazione IntelliJ / Webstorm

1. Assicurati che il plugin NodeJS sia abilitato
2. Seleziona le configurazioni della corsa (schermo)



3. Seleziona **+** > **Debug remoto Node.js**



4. Assicurati di inserire la porta selezionata sopra e l'host corretto

Una volta configurati, esegui semplicemente il target di debug come faresti normalmente e si fermerà sui tuoi breakpoint.

Usa il proxy per il debug tramite la porta su Linux

Se avvii la tua applicazione su Linux, usa il proxy per il debug via porta, ad esempio:

```
socat TCP-LISTEN:9958,fork TCP:127.0.0.1:5858 &
```

Utilizzare quindi la porta 9958 per il debug remoto.

Leggi Debug remoto in Node.JS online: https://riptutorial.com/it/node-js/topic/6335/debug-remoto-in-node-js

Capitolo 26: Design API restful: best practice

Examples

Gestione degli errori: OTTENERE tutte le risorse

Come gestisci gli errori, invece di collegarli alla console?

Cattiva strada:

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        res.json(r)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    });
    request.info.user = req.user._id;
    console.log("ABOUT TO SAVE REQUEST", request);
    request.save((err, r) => {
      if (err) {
        res.json({ message: 'there was an error saving your r' });
      } else {
        res.json(r);
      }
    });
  });
```

Modo migliore:

```
Router.route('/')
 .get((req, res) => {
  Request.find((err, r) => {
   if(err){
    console.log(err)
  } else {
    return next(err)
  }
})
})
.post((req, res) => {
const request = new Request({
  type: req.body.type,
  info: req.body.info
});
request.info.user = req.user._id;
console.log("ABOUT TO SAVE REQUEST", request);
```

```
request.save((err, r) => {
  if (err) {
    return next(err)
  } else {
    res.json(r);
  }
});
});
```

Leggi Design API restful: best practice online: https://riptutorial.com/it/node-js/topic/6490/design-api-restful--best-practice

Capitolo 27: Disinstallazione di Node.js

Examples

Disinstallare completamente Node.js su Mac OSX

In Terminale sul tuo sistema operativo Mac, inserisci i seguenti 2 comandi:

```
lsbom -f -l -s -pf /var/db/receipts/org.nodejs.pkg.bom | while read f; do  sudo rm
/usr/local/${f}; done

sudo rm -rf /usr/local/lib/node /usr/local/lib/node_modules /var/db/receipts/org.nodejs.*
```

Disinstallare Node.js su Windows

Per disinstallare Node.js su Windows, utilizzare Aggiungi o Rimuovi programmi come questo:

1. Apri `Add or Remove Programs` dal menu di avvio.
2. Cerca `Node.js`

Windows 10:

3. Fai clic su Node.js.
4. Clicca su Disinstalla.
5. Fai clic sul nuovo pulsante Disinstalla.

Windows 7-8.1:

3. Fai clic sul pulsante Disinstalla sotto Node.js.

Leggi Disinstallazione di Node.js online: https://riptutorial.com/it/node-js/topic/2821/disinstallazione-di-node-js

Capitolo 28: Distribuzione dell'applicazione Node.js senza tempi di inattività.

Examples

Distribuzione utilizzando PM2 senza tempi di fermo.

`ecosystem.json`

```
{
    "name": "app-name",
    "script": "server",
    "exec_mode": "cluster",
    "instances": 0,
    "wait_ready": true
    "listen_timeout": 10000,
    "kill_timeout": 5000,
}
```

```
wait_ready
```

Invece di ricaricare in attesa dell'evento di ascolto, attendere process.send ('ready');

```
listen_timeout
```

Tempo in ms prima di forzare un ricaricamento se l'app non è in ascolto.

```
kill_timeout
```

Tempo in ms prima di inviare un SIGKLL finale.

`server.js`

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

const app = express();
const server = http.Server(app);
const port = 80;

server.listen(port, function() {
    process.send('ready');
});

process.on('SIGINT', function() {
    server.close(function() {
        process.exit(0);
    });
});
```

Potrebbe essere necessario attendere che l'applicazione abbia connessioni stabilite con i propri DB / cache / worker / qualunque. PM2 deve attendere prima di considerare l'applicazione come online. Per fare ciò, è necessario fornire `wait_ready: true` in un file di processo. Questo renderà PM2 in ascolto per quell'evento. Nella tua applicazione dovrai aggiungere `process.send('ready');` quando vuoi che la tua domanda sia considerata pronta.

Quando un processo viene arrestato / riavviato da PM2, alcuni segnali di sistema vengono inviati al processo in un determinato ordine.

Prima un `SIGINT` viene inviato un segnale ai tuoi processi, segnale che puoi prendere per sapere che il tuo processo verrà fermato. Se la tua applicazione non esce prima di 1.6s (personalizzabile) riceverà un segnale `SIGKILL` per forzare l'uscita del processo. Quindi se la tua applicazione ha bisogno di ripulire qualcosa di stato o di lavoro, puoi prendere il segnale `SIGINT` per preparare la tua richiesta di uscita.

Leggi Distribuzione dell'applicazione Node.js senza tempi di inattività. online: https://riptutorial.com/it/node-js/topic/9752/distribuzione-dell-applicazione-node-js-senza-tempi-di-inattivita-

Capitolo 29: Distribuzione di applicazioni Node.js in produzione

Examples

Impostazione NODE_ENV = "produzione"

Le distribuzioni di produzione variano in molti modi, ma una convenzione standard quando si distribuisce in produzione consiste nel definire una variabile d'ambiente chiamata `NODE_ENV` e impostare il suo valore su *"produzione"* .

Bandiere di runtime

Qualsiasi codice in esecuzione nella tua applicazione (inclusi i moduli esterni) può controllare il valore di `NODE_ENV` :

```
if(process.env.NODE_ENV === 'production') {
    // We are running in production mode
} else {
   // We are running in development mode
}
```

dipendenze

Quando la variabile di ambiente `NODE_ENV` è impostata su *"produzione"*, tutte le `devDependencies` nel file *package.json* verranno completamente ignorate durante l'esecuzione `npm install` . Puoi anche applicarlo con un flag `--production` :

```
npm install --production
```

Per impostare `NODE_ENV` puoi utilizzare uno di questi metodi

metodo 1: imposta NODE_ENV per tutte le app nodo

Finestre :

```
set NODE_ENV=production
```

Linux o altri sistemi basati su Unix:

```
export NODE_ENV=production
```

Questo imposta `NODE_ENV` per la sessione bash corrente, quindi tutte le app avviate dopo questa

istruzione avranno `NODE_ENV` impostato sulla `production` .

metodo 2: imposta NODE_ENV per l'app corrente

```
NODE_ENV=production node app.js
```

Questo imposterà `NODE_ENV` per l'app corrente. Questo aiuta quando vogliamo testare le nostre app in diversi ambienti.

metodo 3: crea il file `.env` e `.env`

Questo usa l'idea spiegata qui . Fare riferimento a questo post per una spiegazione più dettagliata.

Fondamentalmente si crea `.env` file `.env` ed esegui alcuni script bash per impostarli sull'ambiente.

Per evitare di scrivere uno script bash, è possibile utilizzare il pacchetto env-cmd per caricare le variabili d'ambiente definite nel file `.env` .

```
env-cmd .env node app.js
```

metodo 4: utilizzare il pacchetto `cross-env`

Questo pacchetto consente di impostare le variabili di ambiente in un modo per ogni piattaforma.

Dopo averlo installato con npm, puoi aggiungerlo allo script di distribuzione in `package.json` come segue:

```
"build:deploy": "cross-env NODE_ENV=production webpack"
```

Gestisci l'app con il gestore dei processi

È buona norma eseguire app NodeJS controllate da process manager. Process Manager aiuta a mantenere l'applicazione in vita per sempre, ricomincia dall'errore, ricarica senza tempi di fermo e semplifica l'amministrazione. I più potenti di questi (come PM2) hanno un sistema di bilanciamento del carico integrato. PM2 consente inoltre di gestire la registrazione delle applicazioni, il monitoraggio e il clustering.

PM2 Process Manager

Installazione di PM2:

```
npm install pm2 -g
```

Il processo può essere avviato in modalità cluster che prevede il bilanciamento del carico integrato per distribuire il carico tra i processi:

`pm2 start app.js -i 0 --name "api"` (*-i* serve per specificare il numero di processi da generare. Se è 0, il numero di processo sarà basato sul numero di core della CPU)

Pur avendo più utenti in produzione, è necessario disporre di un singolo punto per PM2. Pertanto, il comando pm2 deve essere preceduto da una posizione (per la configurazione PM2) altrimenti verrà generato un nuovo processo pm2 per ogni utente con configurazione nella rispettiva directory principale. E sarà incoerente.

Utilizzo: `PM2_HOME=/etc/.pm2 pm2 start app.js`

Distribuzione utilizzando PM2

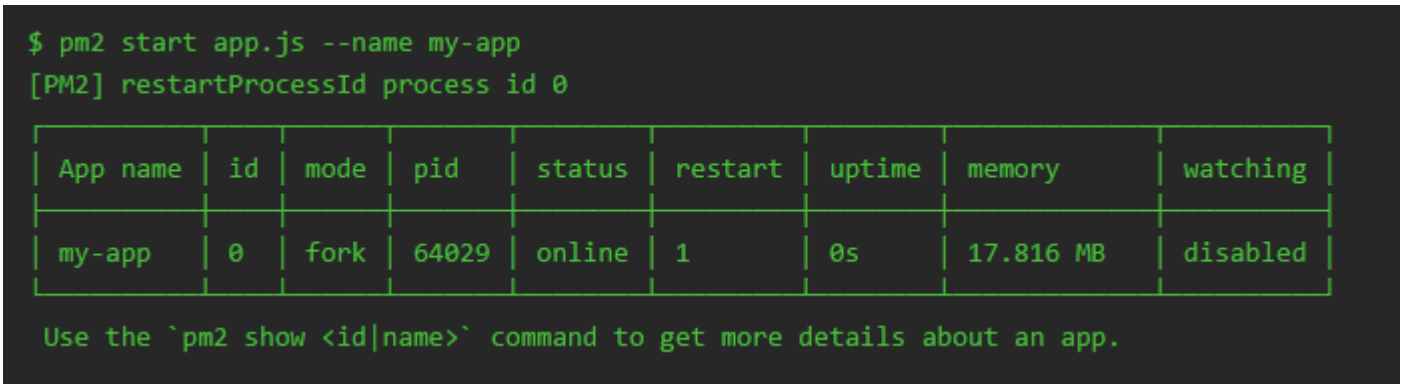
`PM2` è un gestore dei processi di produzione per le applicazioni `Node.js` , che consente di mantenere attive le applicazioni per sempre e di ricaricarle senza tempi di fermo. PM2 consente inoltre di gestire la registrazione delle applicazioni, il monitoraggio e il clustering.

Installa `pm2` livello globale.

```
npm install -g pm2
```

Quindi, esegui l'app `node.js` utilizzando `PM2` .

```
pm2 start server.js --name "my-app"
```



I seguenti comandi sono utili mentre si lavora con `PM2` .

Elenca tutti i processi in esecuzione:

```
pm2 list
```

Interrompi un'app:

```
pm2 stop my-app
```

Riavvia un'app:

```
pm2 restart my-app
```

Per visualizzare informazioni dettagliate su un'app:

```
pm2 show my-app
```

Per rimuovere un'app dal registro di PM2:

```
pm2 delete my-app
```

Distribuzione utilizzando il gestore dei processi

Generalmente, il process manager viene utilizzato in produzione per distribuire un'app nodejs. Le funzioni principali di un gestore processi stanno riavviando il server in caso di arresto anomalo, controllo del consumo delle risorse, miglioramento delle prestazioni di runtime, monitoraggio, ecc.

Alcuni dei process manager popolari realizzati dalla comunità dei nodi sono per sempre, pm2, ecc.

Forvever

`forever` è uno strumento di interfaccia a riga di comando per garantire che un determinato script venga eseguito continuamente. La semplice interfaccia di `forever` lo rende ideale per eseguire piccole distribuzioni di app e script `Node.js`

monitora `forever` tuo processo e lo riavvia se si blocca.

Installa `forever` livello globale.

```
$ npm install -g forever
```

Esegui l'applicazione:

```
$ forever start server.js
```

Questo avvia il server e fornisce un id per il processo (parte da 0).

Riavvia l'applicazione:

```
$ forever restart 0
```

Qui `0` è l'id del server.

Arresta l'applicazione:

```
$ forever stop 0
```

Simile al riavvio, `0` è l'id del server. Puoi anche fornire ID processo o nome script al posto dell'ID dato da sempre.

Per ulteriori comandi: https://www.npmjs.com/package/forever

Utilizzo di diverse proprietà / configurazione per ambienti diversi come dev, qa, staging, ecc.

Le applicazioni su larga scala spesso hanno bisogno di proprietà diverse quando funzionano su ambienti diversi. possiamo ottenere ciò passando argomenti all'applicazione NodeJs e utilizzando lo stesso argomento nel processo del nodo per caricare un file di proprietà dell'ambiente specifico.

Supponiamo di avere due file di proprietà per diversi ambienti.

- dev.json

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

- qa.json

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

Il codice seguente nell'applicazione esporterà il rispettivo file di proprietà che vogliamo utilizzare.

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

Diamo argomenti all'applicazione come seguendo

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

se usiamo process manager come *per sempre* che sia semplice come

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

Approfittando dei cluster

Una singola istanza di Node.js viene eseguita in un singolo thread. Per sfruttare i sistemi multi-core, l'utente a volte desidera avviare un cluster di processi Node.js per gestire il carico.

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

   var numCPUs = require('os').cpus().length;

       if (cluster.isMaster) {
         // In real life, you'd probably use more than just 2 workers,
         // and perhaps not put the master and worker in the same file.
         //
         // You can also of course get a bit fancier about logging, and
         // implement whatever custom logic you need to prevent DoS
         // attacks and other bad behavior.
         //
         // See the options in the cluster documentation.
         //
         // The important thing is that the master does very little,
         // increasing our resilience to unexpected errors.
           console.log('your server is working on ' + numCPUs + ' cores');

            for (var i = 0; i < numCPUs; i++) {
                   cluster.fork();
                 }

         cluster.on('disconnect', function(worker) {
           console.error('disconnect!');
           //clearTimeout(timeout);
           cluster.fork();
         });


       } else {
           require('./app.js');

       }
```

Leggi Distribuzione di applicazioni Node.js in produzione online: https://riptutorial.com/it/node-js/topic/2975/distribuzione-di-applicazioni-node-js-in-produzione

Capitolo 30: ECMAScript 2015 (ES6) con Node.js

Examples

const / let dichiarazioni

A differenza di `var` , `const` / `let` sono legati all'ambito lessicale piuttosto che all'ambito della funzione.

```
{
  var x = 1 // will escape the scope
  let y = 2 // bound to lexical scope
  const z = 3 // bound to lexical scope, constant
}

console.log(x) // 1
console.log(y) // ReferenceError: y is not defined
console.log(z) // ReferenceError: z is not defined
```

Esegui in RunKit

Funzioni della freccia

Le funzioni della freccia si associano automaticamente all'ambito lessicale "this" del codice circostante.

```
performSomething(result => {
  this.someVariable = result
})
```

vs

```
performSomething(function(result) {
  this.someVariable = result
}.bind(this))
```

Esempio di funzione freccia

Consideriamo questo esempio, che emette i quadrati dei numeri 3, 5 e 7:

```
let nums = [3, 5, 7]
let squares = nums.map(function (n) {
  return n * n
})
console.log(squares)
```

Esegui in RunKit

La funzione passata a `.map` può anche essere scritta come funzione di freccia rimuovendo la parola chiave `function` e aggiungendo invece la freccia `=>` :

```
let nums = [3, 5, 7]
let squares = nums.map((n) => {
  return n * n
})
console.log(squares)
```

Esegui in RunKit

Tuttavia, questo può essere scritto in modo ancora più conciso. Se il corpo della funzione è costituito da una sola istruzione e tale istruzione calcola il valore restituito, è possibile rimuovere le parentesi graffe di avvolgere il corpo della funzione e la parola chiave `return` .

```
let nums = [3, 5, 7]
let squares = nums.map(n => n * n)
console.log(squares)
```

Esegui in RunKit

destrutturazione

```
    let [x,y, ...nums] = [0, 1, 2, 3, 4, 5, 6];
console.log(x, y, nums);

let {a, b, ...props} = {a:1, b:2, c:3, d:{e:4}}
console.log(a, b, props);

let dog = {name: 'fido', age: 3};
let {name:n, age} = dog;
console.log(n, age);
```

flusso

```
/* @flow */

function product(a: number, b: number){
  return a * b;
}

const b = 3;
let c = [1,2,3,,{}];
let d = 3;

import request from 'request';

request('http://dev.markitondemand.com/MODApis/Api/v2/Quote/json?symbol=AAPL', (err, res,
payload)=>{
  payload = JSON.parse(payload);
  let {LastPrice} = payload;
  console.log(LastPrice);
```

```
});
```

Classe ES6

```
class Mammel {
  constructor(legs){
    this.legs = legs;
  }
  eat(){
    console.log('eating...');
  }
  static count(){
    console.log('static count...');
  }
}

class Dog extends Mammel{
  constructor(name, legs){
    super(legs);
    this.name = name;
  }
  sleep(){
    super.eat();
    console.log('sleeping');
  }
}

let d = new Dog('fido', 4);
d.sleep();
d.eat();
console.log('d', d);
```

Leggi ECMAScript 2015 (ES6) con Node.js online: https://riptutorial.com/it/node-js/topic/6732/ecmascript-2015--es6--con-node-js

Capitolo 31: Emettitori di eventi

Osservazioni

Quando un evento "spara" (che equivale a "pubblicare un evento" o "emettere un evento"), ciascun listener verrà chiamato in modo sincrono (sorgente), insieme ai dati di accompagnamento che sono stati passati a `emit()` , no importa quanti argomenti passi in:

```
myDog.on('bark', (howLoud, howLong, howIntense) => {
  // handle the event
})
myDog.emit('bark', 'loudly', '5 seconds long', 'fiercely')
```

Gli ascoltatori verranno chiamati nell'ordine in cui sono stati registrati:

```
myDog.on('urinate', () => console.log('My first thought was "Oh-no"'))
myDog.on('urinate', () => console.log('My second thought was "Not my lawn :)"'))
myDog.emit('urinate')
// The console.logs will happen in the right order because they were registered in that order.
```

Ma se hai bisogno di un ascoltatore per sparare prima, prima di tutti gli altri listener che sono già stati aggiunti, puoi usare `prependListener()` modo:

```
myDog.prependListener('urinate', () => console.log('This happens before my first and second
thoughts, even though it was registered after them'))
```

Se hai bisogno di ascoltare un evento, ma vuoi sentirlo solo una volta, puoi usare `once` invece di `on` o `prependOnceListener` invece di `prependListener` . Dopo che l'evento è stato attivato e il listener viene chiamato, il listener verrà automaticamente rimosso e non verrà richiamato la volta successiva che l'evento viene attivato.

Infine, se vuoi rimuovere tutti gli ascoltatori e ricominciare da capo, sentiti libero di fare proprio questo:

```
myDog.removeAllListeners()
```

Examples

Analisi HTTP tramite un emettitore di eventi

Nel codice del server HTTP (es. `server.js`):

```
const EventEmitter = require('events')
const serverEvents = new EventEmitter()

// Set up an HTTP server
const http = require('http')
```

```
const httpServer = http.createServer((request, response) => {
  // Handler the request...
  // Then emit an event about what happened
  serverEvents.emit('request', request.method, request.url)
});

// Expose the event emitter
module.exports = serverEvents
```

Nel codice supervisore (es. `supervisor.js`):

```
const server = require('./server.js')
// Since the server exported an event emitter, we can listen to it for changes:
server.on('request', (method, url) => {
  console.log(`Got a request: ${method} ${url}`)
})
```

Ogni volta che il server riceve una richiesta, emetterà un evento chiamato `request` che il supervisore sta ascoltando, e quindi il supervisore può reagire all'evento.

Nozioni di base

Gli emettitori di eventi sono incorporati nel nodo e sono per pub-sub, un modello in cui un *editore* emetterà eventi, a cui gli *abbonati* possono ascoltare e reagire. In gergo Node, gli editori sono chiamati *Emettitori di eventi* e emettono eventi, mentre gli abbonati sono chiamati *ascoltatori* e reagiscono agli eventi.

```
// Require events to start using them
const EventEmitter = require('events').EventEmitter;
// Dogs have events to publish, or emit
class Dog extends EventEmitter {};
class Food {};

let myDog = new Dog();

// When myDog is chewing, run the following function
myDog.on('chew', (item) => {
  if (item instanceof Food) {
    console.log('Good dog');
  } else {
    console.log(`Time to buy another ${item}`);
  }
});

myDog.emit('chew', 'shoe'); // Will result in console.log('Time to buy another shoe')
const bacon = new Food();
myDog.emit('chew', bacon); // Will result in console.log('Good dog')
```

Nell'esempio precedente, il cane è l'editore / EventEmitter, mentre la funzione che controlla l'elemento è l'utente / ascoltatore. Puoi anche fare più ascoltatori:

```
myDog.on('bark', () => {
  console.log('WHO\'S AT THE DOOR?');
  // Panic
```

```
});
```

Possono anche esserci più ascoltatori per un singolo evento e persino rimuovere gli ascoltatori:

```
myDog.on('chew', takeADeepBreathe);
myDog.on('chew', calmDown);
// Undo the previous line with the next one:
myDog.removeListener('chew', calmDown);
```

Se vuoi ascoltare un evento solo una volta, puoi usare:

```
myDog.once('chew', pet);
```

Che rimuoverà l'ascoltatore automaticamente senza condizioni di gara.

Ottieni i nomi degli eventi a cui sei iscritto

La funzione **EventEmitter.eventNames ()** restituirà un array contenente i nomi degli eventi attualmente sottoscritti.

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}

var emitter = new MyEmitter();

emitter
.on("message", function(){ //listen for message event
    console.log("a message was emitted!");
})
.on("message", function(){ //listen for message event
    console.log("this is not the right message");
})
.on("data", function(){ //listen for data event
    console.log("a data just occured!!");
});

console.log(emitter.eventNames()); //=> ["message","data"]
emitter.removeAllListeners("data");//=> removeAllListeners to data event
console.log(emitter.eventNames()); //=> ["message"]
```

Esegui in RunKit

Ottieni il numero di ascoltatori registrati per ascoltare un evento specifico

La funzione Emitter.listenerCount (eventName) restituirà il numero di listener che sono attualmente in ascolto per l'evento fornito come argomento

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}
var emitter = new MyEmitter();

emitter
.on("data", ()=>{ // add listener for data event
```

```
    console.log("data event emitter");
});

console.log(emitter.listenerCount("data"))    // => 1
console.log(emitter.listenerCount("message")) // => 0

emitter.on("message", function mListener(){ //add listener for message event
    console.log("message event emitted");
});
console.log(emitter.listenerCount("data"))    // => 1
console.log(emitter.listenerCount("message")) // => 1

emitter.once("data", (stuff)=>{ //add another listener for data event
    console.log(`Tell me my ${stuff}`);
})

console.log(emitter.listenerCount("data"))   // => 2
console.log(emitter.listenerCount("message"))// => 1
```

Leggi Emettitori di eventi online: https://riptutorial.com/it/node-js/topic/1623/emettitori-di-eventi

Capitolo 32: Esecuzione di file o comandi con Child Processes

Sintassi

- child_process.exec (comando [, opzioni] [, callback])
- child_process.execFile (file [, args] [, opzioni] [, callback])
- child_process.fork (modulePath [, args] [, options])
- child_process.spawn (command [, args] [, options])
- child_process.execFileSync (file [, args] [, opzioni])
- child_process.execSync (comando [, opzioni])
- child_process.spawnSync (command [, args] [, options])

Osservazioni

Quando si `ChildProcess` processi figli, tutti i metodi asincroni restituiranno un'istanza di `ChildProcess` , mentre tutte le versioni sincrone restituiranno l'output di qualunque cosa sia stata eseguita. Come altre operazioni sincrone in Node.js, se si verifica un errore, *verrà* generato.

Examples

Creazione di un nuovo processo per eseguire un comando

Per generare un nuovo processo in cui è necessario un output *non bufferizzato* (ad esempio, processi a esecuzione prolungata che potrebbero stampare l'output in un determinato periodo di tempo anziché stampare ed uscire immediatamente), utilizzare `child_process.spawn()` .

Questo metodo genera un nuovo processo utilizzando un dato comando e un array di argomenti. Il valore restituito è un'istanza di `ChildProcess` , che a sua volta fornisce le proprietà `stdout` e `stderr` . Entrambi questi flussi sono istanze di `stream.Readable` . `stream.Readable` .

Il seguente codice equivale a utilizzare il comando `ls -lh /usr` .

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});

ls.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
```

```
});
```

Un altro comando di esempio:

```
zip -0vr "archive" ./image.png
```

Potrebbe essere scritto come:

```
spawn('zip', ['-0vr', '"archive"', './image.png']);
```

Creazione di una shell per eseguire un comando

Per eseguire un comando in una shell, in cui è richiesto l'output bufferizzato (ovvero non è un flusso), utilizzare `child_process.exec` . Ad esempio, se si desidera eseguire il comando `cat *.js file | wc -l` , senza opzioni, sarebbe simile a questo:

```
const exec = require('child_process').exec;
exec('cat *.js file | wc -l', (err, stdout, stderr) => {
  if (err) {
    console.error(`exec error: ${err}`);
    return;
  }

  console.log(`stdout: ${stdout}`);
  console.log(`stderr: ${stderr}`);
});
```

La funzione accetta fino a tre parametri:

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

Il parametro command è una stringa ed è obbligatorio, mentre l'oggetto options e callback sono entrambi facoltativi. Se non viene specificato alcun oggetto opzioni, `exec` userà quanto segue come predefinito:

```
{
  encoding: 'utf8',
  timeout: 0,
  maxBuffer: 200*1024,
  killSignal: 'SIGTERM',
  cwd: null,
  env: null
}
```

L'oggetto options supporta anche un parametro `shell` , che è di default `/bin/sh` su UNIX e `cmd.exe` su Windows, un'opzione `uid` per l'impostazione dell'identità utente del processo e un'opzione `gid` per l'identità di gruppo.

Il callback, che viene chiamato quando viene eseguito il comando, viene chiamato con i tre argomenti `(err, stdout, stderr)` . Se il comando viene eseguito correttamente, `err` sarà `null` ,

altrimenti sarà un'istanza di `Error` , dove `err.code` è il codice di uscita del processo e `err.signal` è il segnale che è stato inviato per terminarlo.

Gli argomenti `stdout` e `stderr` sono l'output del comando. Viene decodificato con la codifica specificata nell'oggetto options (default: `string`), ma può essere altrimenti restituito come oggetto `Buffer` .

Esiste anche una versione sincrona di `exec` , che è `execSync` . La versione sincrona non `ChildProcess` una richiamata e restituirà `stdout` anziché un'istanza di `ChildProcess` . Se la versione sincrona incontra un errore, il programma *verrà* lanciato e interrotto. Sembra questo:

```
const execSync = require('child_process').execSync;
const stdout = execSync('cat *.js file | wc -l');
console.log(`stdout: ${stdout}`);
```

Generare un processo per eseguire un eseguibile

Se stai cercando di eseguire un file, come un eseguibile, usa `child_process.execFile` . Invece di `child_process.exec` una shell come `child_process.exec` , creerà direttamente un nuovo processo, che è leggermente più efficiente di un comando. La funzione può essere utilizzata in questo modo:

```
const execFile = require('child_process').execFile;
const child = execFile('node', ['--version'], (err, stdout, stderr) => {
  if (err) {
    throw err;
  }

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

A differenza di `child_process.exec` , questa funzione accetta fino a quattro parametri, in cui il secondo parametro è un array di argomenti che desideri fornire all'eseguibile:

```
child_process.execFile(file[, args][, options][, callback]);
```

Altrimenti, le opzioni e il formato di callback sono altrimenti identici a `child_process.exec` . Lo stesso vale per la versione sincrona della funzione:

```
const execFileSync = require('child_process').execFileSync;
const stdout = execFileSync('node', ['--version']);
console.log(stdout);
```

Leggi Esecuzione di file o comandi con Child Processes online: https://riptutorial.com/it/node-js/topic/2726/esecuzione-di-file-o-comandi-con-child-processes

Capitolo 33: Esecuzione di node.js come servizio

introduzione

A differenza di molti server Web, Node non è installato come un servizio pronto all'uso. Ma in produzione, è meglio farlo funzionare come un demone, gestito da un sistema di init.

Examples

Node.js come un demone di sistema

systemd è il sistema di init di *fatto* nella maggior parte delle distribuzioni Linux. Dopo che il nodo è stato configurato per l'esecuzione con systemd, è possibile utilizzare il comando di `service` per gestirlo.

Prima di tutto, ha bisogno di un file di configurazione, creiamo. Per le distro basate su Debian, sarà in `/etc/systemd/system/node.service`

```
[Unit]
Description=My super nodejs app

[Service]
# set the working directory to have consistent relative paths
WorkingDirectory=/var/www/app

# start the server file (file is relative to WorkingDirectory here)
ExecStart=/usr/bin/node serverCluster.js

# if process crashes, always try to restart
Restart=always

# let 500ms between the crash and the restart
RestartSec=500ms

# send log tot syslog here (it doesn't compete with other log config in the app itself)
StandardOutput=syslog
StandardError=syslog

# nodejs process name in syslog
SyslogIdentifier=nodejs

# user and group starting the app
User=www-data
Group=www-data

# set the environement (dev, prod…)
Environment=NODE_ENV=production


[Install]
```

```
# start node at multi user system level (= sysVinit runlevel 3)
WantedBy=multi-user.target
```

Ora è possibile avviare, arrestare e riavviare l'app rispettivamente con:

```
service node start
service node stop
service node restart
```

Per dire a systemd di avviare automaticamente il nodo all'avvio, basta digitare: `systemctl enable node` .

Questo è tutto, il nodo ora funziona come un demone.

Leggi Esecuzione di node.js come servizio online: https://riptutorial.com/it/node-js/topic/9258/esecuzione-di-node-js-come-servizio

Capitolo 34: Esportazione e consumo di moduli

Osservazioni

Mentre tutto in Node.js viene generalmente eseguito in modo asincrono, `require()` non è una di quelle cose. Poiché in pratica i moduli devono essere caricati solo una volta, si tratta di un'operazione di blocco e devono essere utilizzati correttamente.

I moduli vengono memorizzati nella cache dopo la prima volta che vengono caricati. Se si sta modificando un modulo in fase di sviluppo, sarà necessario cancellare la sua voce nella cache del modulo per poter utilizzare le nuove modifiche. Detto questo, anche se un modulo viene cancellato dalla cache del modulo, il modulo stesso non viene raccolto, quindi è necessario prestare attenzione per il suo utilizzo negli ambienti di produzione.

Examples

Caricamento e utilizzo di un modulo

Un modulo può essere "importato" o altrimenti "richiesto" dalla funzione `require()` . Ad esempio, per caricare il modulo `http` fornito con Node.js, è possibile utilizzare quanto segue:

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

Oltre ai moduli forniti con il runtime, è anche possibile richiedere i moduli installati da npm, ad esempio express. Se avevi già installato Express sul tuo sistema tramite `npm install express` , potresti semplicemente scrivere:

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

Puoi anche includere moduli scritti da te come parte della tua applicazione. In questo caso, per includere un file denominato `lib.js` nella stessa directory del file corrente:

```
const mylib = require('./lib');
```

Nota che puoi omettere l'estensione e si supporrà `.js` . Una volta caricato un modulo, la variabile viene popolata con un oggetto che contiene i metodi e le proprietà pubblicati dal file richiesto. Un esempio completo:

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

// The `http` module has the property `STATUS_CODES`
console.log(http.STATUS_CODES[404]); // outputs 'Not Found'
```

```
// Also contains `createServer()`
http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<html><body>Module Test</body></html>');
  res.end();
}).listen(80);
```

Creazione di un modulo hello-world.js

Il nodo fornisce l'interfaccia `module.exports` per esporre funzioni e variabili ad altri file. Il modo più semplice per farlo è esportare solo un oggetto (funzione o variabile), come mostrato nel primo esempio.

Hello-world.js

```
module.exports = function(subject) {
    console.log('Hello ' + subject);
};
```

Se non vogliamo che l'intero export sia un singolo oggetto, possiamo esportare funzioni e variabili come proprietà dell'oggetto `exports` . I tre esempi seguenti dimostrano tutto ciò in modi leggermente diversi:

- hello-venus.js: la definizione della funzione viene eseguita separatamente, quindi aggiunta come proprietà di `module.exports`
- ciao-jupiter.js: le definizioni delle funzioni sono messe direttamente come valore delle proprietà di `module.exports`
- hello-mars.js: la definizione della funzione viene dichiarata direttamente come una proprietà di `exports` che è una versione breve di `module.exports`

ciao-venus.js

```
function hello(subject) {
    console.log('Venus says Hello ' + subject);
}

module.exports = {
    hello: hello
};
```

Hello-jupiter.js

```
module.exports = {
    hello: function(subject) {
      console.log('Jupiter says hello ' + subject);
    },

    bye: function(subject) {
      console.log('Jupiter says goodbye ' + subject);
    }
};
```

Hello-mars.js

```
exports.hello = function(subject) {
    console.log('Mars says Hello ' + subject);
};
```

Caricamento del modulo con il nome della directory

Abbiamo una directory chiamata `hello` che include i seguenti file:

index.js

```
// hello/index.js
module.exports = function(){
    console.log('Hej');
};
```

main.js

```
// hello/main.js
// We can include the other files we've defined by using the `require()` method
var hw = require('./hello-world.js'),
    hm = require('./hello-mars.js'),
    hv = require('./hello-venus.js'),
    hj = require('./hello-jupiter.js'),
    hu = require('./index.js');

// Because we assigned our function to the entire `module.exports` object, we
// can use it directly
hw('World!'); // outputs "Hello World!"

// In this case, we assigned our function to the `hello` property of exports, so we must
// use that here too
hm.hello('Solar System!'); // outputs "Mars says Hello Solar System!"

// The result of assigning module.exports at once is the same as in hello-world.js
hv.hello('Milky Way!'); // outputs "Venus says Hello Milky Way!"

hj.hello('Universe!'); //  outputs "Jupiter says hello Universe!"
hj.bye('Universe!'); // outputs "Jupiter says goodbye Universe!"

hu(); //output 'hej'
```

Invalida della cache del modulo

In fase di sviluppo, potresti scoprire che l'uso di `require()` sullo stesso modulo più volte restituisce sempre lo stesso modulo, anche se hai apportato modifiche a quel file. Questo perché i moduli vengono memorizzati nella cache la prima volta che vengono caricati e tutti i successivi carichi del modulo verranno caricati dalla cache.

Per ovviare a questo problema, dovrai `delete` la voce nella cache. Ad esempio, se hai caricato un modulo:

```
var a = require('./a');
```

È quindi possibile eliminare la voce della cache:

```
var rpath = require.resolve('./a.js');
delete require.cache[rpath];
```

E quindi richiedi di nuovo il modulo:

```
var a = require('./a');
```

Si noti che questo non è raccomandato in produzione perché l' `delete` cancellerà solo il riferimento al modulo caricato, non i dati caricati stessi. Il modulo non è garbage collection, quindi l'uso improprio di questa funzione potrebbe portare alla perdita di memoria.

Costruire i tuoi moduli

Puoi anche fare riferimento a un oggetto per esportare pubblicamente e aggiungere continuamente metodi a quell'oggetto:

```
const auth = module.exports = {}
const config = require('../config')
const request = require('request')

auth.email = function (data, callback) {
  // Authenticate with an email address
}

auth.facebook = function (data, callback) {
  // Authenticate with a Facebook account
}

auth.twitter = function (data, callback) {
  // Authenticate with a Twitter account
}

auth.slack = function (data, callback) {
  // Authenticate with a Slack account
}

auth.stack_overflow = function (data, callback) {
  // Authenticate with a Stack Overflow account
}
```

Per utilizzare uno di questi, basta richiedere il modulo come faresti normalmente:

```
const auth = require('./auth')

module.exports = function (req, res, next) {
  auth.facebook(req.body, function (err, user) {
    if (err) return next(err)

    req.user = user
    next()
```

```
  })
}
```

Ogni modulo iniettato una sola volta

NodeJS esegue il modulo solo la prima volta che lo richiedi. Ogni ulteriore richiesta di funzioni riutilizzerà lo stesso oggetto, quindi non eseguirà il codice nel modulo un'altra volta. Anche il nodo memorizza nella cache i moduli la prima volta che vengono caricati utilizzando require. Ciò riduce il numero di letture di file e aiuta a velocizzare l'applicazione.

myModule.js

```
console.log(123) ;
exports.var1 = 4 ;
```

index.js

```
var a=require('./myModule') ; // Output 123
var b=require('./myModule') ; // No output
console.log(a.var1) ; // Output 4
console.log(b.var1) ; // Output 4
a.var2 = 5 ;
console.log(b.var2) ; // Output 5
```

Caricamento del modulo da node_modules

I moduli possono essere `require` d senza utilizzare percorsi relativi inserendoli in una directory speciale chiamata `node_modules` .

Ad esempio, per `require` un modulo chiamato `foo` da un file `index.js` , è possibile utilizzare la seguente struttura di directory:

```
index.js
 \- node_modules
  \- foo
   |- foo.js
   \- package.json
```

I moduli dovrebbero essere collocati all'interno di una directory, insieme a un file `package.json` . Il campo `main` del file `package.json` dovrebbe puntare al punto di ingresso per il tuo modulo - questo è il file che viene importato quando gli utenti lo `require('your-module')` . `main` default `index.js` se non previsto. In alternativa, è possibile fare riferimento a file in relazione al modulo semplicemente aggiungendo il percorso relativo alla `require` chiamata: `require('your-module/path/to/file')` .

I moduli possono anche `require` d dalle directory `node_modules` fino alla gerarchia del file system. Se abbiamo la seguente struttura di directory:

```
my-project
\- node_modules
 |- foo   // the foo module
```

```
  \- ...
 \- baz   // the baz module
  \- node_modules
   \- bar   // the bar module
```

saremo in grado di `require` il modulo `foo` da qualsiasi file all'interno della `bar` usando `require('foo')` .

Si noti che il nodo abbinerà solo il modulo più vicino al file nella gerarchia del filesystem, a partire da (la directory corrente del file / node_modules). Nodo corrisponde alle directory in questo modo fino alla radice del file system.

È possibile installare nuovi moduli dal registro di npm o altri registri di npm o crearne di nuovi.

Cartella come modulo

I moduli possono essere suddivisi su molti file .js nella stessa cartella. Un esempio in una cartella *my_module* :

function_one.js

```
module.exports = function() {
  return 1;
}
```

function_two.js

```
module.exports = function() {
  return 2;
}
```

index.js

```
exports.f_one = require('./function_one.js');
exports.f_two = require('./function_two.js');
```

Un modulo come questo viene utilizzato facendo riferimento ad esso dal nome della cartella:

```
var split_module = require('./my_module');
```

Si noti che se lo si richiede omettendo `./` o qualsiasi indicazione di un percorso di una cartella dall'argomento della funzione require, Node proverà a caricare un modulo dalla cartella *node_modules* .

In alternativa puoi creare nella stessa cartella un file `package.json` con questi contenuti:

```
{
    "name": "my_module",
    "main": "./your_main_entry_point.js"
}
```

In questo modo non è necessario nominare il file del modulo principale "index".

Leggi Esportazione e consumo di moduli online: https://riptutorial.com/it/node-js/topic/547/esportazione-e-consumo-di-moduli

Capitolo 35: Esportazione e importazione del modulo in node.js

Examples

Utilizzando un modulo semplice in node.js

Che cos'è un modulo node.js (link all'articolo):

> Un modulo incapsula codice correlato in una singola unità di codice. Quando si crea un modulo, questo può essere interpretato come lo spostamento di tutte le funzioni correlate in un file.

Ora vediamo un esempio. Immagina che tutti i file siano nella stessa directory:

File: `printer.js`

```
"use strict";

exports.printHelloWorld = function (){
    console.log("Hello World!!!");
}
```

Un altro modo di utilizzare i moduli:

File `animals.js`

```
"use strict";

module.exports = {
    lion: function() {
        console.log("ROAARR!!!");
    }

};
```

File: `app.js`

Esegui questo file andando nella tua directory e digitando: `node app.js`

```
"use strict";

//require('./path/to/module.js') node which module to load
var printer = require('./printer');
var animals = require('./animals');

printer.printHelloWorld(); //prints "Hello World!!!"
animals.lion(); //prints "ROAARR!!!"
```

Uso delle importazioni in ES6

Node.js è costruito contro le versioni moderne di V8. Aggiornandosi con le ultime versioni di questo motore, ci assicuriamo che le nuove funzionalità della specifica JavaScript ECMA-262 vengano portate agli sviluppatori di Node.js in modo tempestivo, così come i continui miglioramenti delle prestazioni e della stabilità.

Tutte le funzioni di ECMAScript 2015 (ES6) sono suddivise in tre gruppi per le funzioni di spedizione, messa in scena e in corso:

Tutte le funzionalità di spedizione, che V8 considera stabili, sono attivate per impostazione predefinita su Node.js e NON richiedono alcun tipo di flag di runtime. Le funzionalità messe in scena, che sono funzionalità quasi completate che non sono considerate stabili dal team V8, richiedono un flag di runtime: --harmony. Le funzioni in corso possono essere attivate singolarmente mediante il rispettivo flag di armonia, sebbene questo sia altamente sconsigliato a meno che non per scopi di test. Nota: questi flag sono esposti da V8 e cambieranno potenzialmente senza alcun avviso di deprecazione.

Attualmente ES6 supporta le istruzioni di importazione in modo nativo. Fare riferimento qui

Quindi se abbiamo un file chiamato `fun.js` ...

```
export default function say(what){
  console.log(what);
}

export function sayLoud(whoot) {
  say(whoot.toUpperCase());
}
```

... e se c'era un altro file chiamato `app.js` cui vogliamo inserire le funzioni precedentemente definite, ci sono tre modi per importarle.

Importa predefinito

```
import say from './fun';
say('Hello Stack Overflow!!');  // Output: Hello Stack Overflow!!
```

Importa la funzione `say()` perché è contrassegnata come l'esportazione predefinita nel file sorgente (`export default …`)

Importazioni con nome

```
import { sayLoud } from './fun';
sayLoud('JS modules are awesome.'); // Output: JS MODULES ARE AWESOME.
```

Le importazioni con nome ci consentono di importare esattamente le parti di un modulo di cui abbiamo effettivamente bisogno. Lo facciamo nominandoli esplicitamente. Nel nostro caso nominando `sayLoud` tra parentesi graffe all'interno della dichiarazione di importazione.

Importazione in bundle

```
import * as i from './fun';
i.say('What?'); // Output: What?
i.sayLoud('Whoot!'); // Output: WHOOT!
```

Se vogliamo avere tutto, questa è la strada da percorrere. Usando la sintassi `* as i` abbiamo la dichiarazione di `import` fornirci un oggetto `i` che contiene tutte le esportazioni del nostro modulo `fun` come proprietà corrispondenti.

percorsi

Tieni presente che devi contrassegnare esplicitamente i percorsi di importazione come percorsi *relativi* anche se il file da importare risiedeva nella stessa directory come il file in cui stai importando utilizzando `./` . Importa da percorsi non prefissati come

```
import express from 'express';
```

verrà cercato nelle cartelle `node_modules` locali e globali e genererà un errore se non viene trovato alcun modulo corrispondente.

Esportazione con sintassi ES6

Questo è l'equivalente dell'altro esempio, ma usando ES6.

```
export function printHelloWorld() {
  console.log("Hello World!!!");
}
```

Leggi Esportazione e importazione del modulo in node.js online: https://riptutorial.com/it/node-js/topic/1173/esportazione-e-importazione-del-modulo-in-node-js

Capitolo 36: Eventloop

introduzione

In questo post discuteremo come è emerso il concetto di Eventloop e come può essere utilizzato per server ad alte prestazioni e applicazioni guidate da eventi come le GUI.

Examples

Come si è evoluto il concetto di event loop.

Eventloop in pseudo codice

Un ciclo di eventi è un ciclo che attende gli eventi e quindi reagisce a quegli eventi

```
while true:
    wait for something to happen
    react to whatever happened
```

Esempio di un server HTTP a thread singolo senza loop di eventi

```
    while true:
    socket = wait for the next TCP connection
    read the HTTP request headers from (socket)
    file_contents = fetch the requested file from disk
    write the HTTP response headers to (socket)
    write the (file_contents) to (socket)
    close(socket)
```

Ecco una semplice forma di un server HTTP che è un singolo thread ma nessun ciclo di eventi. Il problema qui è che attende fino a quando ogni richiesta è terminata prima di iniziare a elaborare il successivo. Se è necessario un po 'di tempo per leggere le intestazioni delle richieste HTTP o per recuperare il file dal disco, dovremmo essere in grado di avviare l'elaborazione della richiesta successiva mentre attendiamo che finisca.

La soluzione più comune è rendere il programma multi-threaded.

Esempio di un server HTTP multi-thread con nessun ciclo di eventi

```
function handle_connection(socket):
    read the HTTP request headers from (socket)
    file_contents = fetch the requested file from disk
    write the HTTP response headers to (socket)
    write the (file_contents) to (socket)
    close(socket)
while true:
    socket = wait for the next TCP connection
    spawn a new thread doing handle_connection(socket)
```

Ora abbiamo reso il nostro piccolo server HTTP multi thread. In questo modo, possiamo passare immediatamente alla richiesta successiva perché la richiesta corrente è in esecuzione in un thread in background. Molti server, incluso Apache, utilizzano questo approccio.

Ma non è perfetto. Una limitazione è che puoi generare solo tanti thread. Per carichi di lavoro in cui si dispone di un numero enorme di connessioni, ma ogni connessione richiede solo l'attenzione di tanto in tanto, il modello multi-thread non funzionerà molto bene. La soluzione per questi casi è utilizzare un ciclo di eventi:

Esempio di un server HTTP con ciclo di eventi

```
while true:
    event = wait for the next event to happen
    if (event.type == NEW_TCP_CONNECTION):
        conn = new Connection
        conn.socket = event.socket
        start reading HTTP request headers from (conn.socket) with userdata = (conn)
    else if (event.type == FINISHED_READING_FROM_SOCKET):
        conn = event.userdata
        start fetching the requested file from disk with userdata = (conn)
    else if (event.type == FINISHED_READING_FROM_DISK):
        conn = event.userdata
        conn.file_contents = the data we fetched from disk
        conn.current_state = "writing headers"
        start writing the HTTP response headers to (conn.socket) with userdata = (conn)
    else if (event.type == FINISHED_WRITING_TO_SOCKET):
        conn = event.userdata
        if (conn.current_state == "writing headers"):
            conn.current_state = "writing file contents"
            start writing (conn.file_contents) to (conn.socket) with userdata = (conn)
        else if (conn.current_state == "writing file contents"):
            close(conn.socket)
```

Speriamo che questo pseudocodice sia comprensibile. Ecco cosa sta succedendo: Aspettiamo che le cose accadano. Ogni volta che viene creata una nuova connessione o una connessione esistente richiede la nostra attenzione, ci occupiamo di essa, quindi torniamo ad aspettare. In questo modo, ci comportiamo bene quando ci sono molte connessioni e ognuna raramente richiede attenzione.

In un'applicazione reale (non pseudocodice) in esecuzione su Linux, la parte "aspetta che

succeda l'evento successivo" sarebbe implementata chiamando la chiamata di sistema poll () o epoll (). Le parti "inizia a leggere / scrivere qualcosa su un socket" sarebbero implementate chiamando le chiamate di sistema recv () o send () in modalità non bloccante.

Riferimento:

[1]. "Come funziona un ciclo di eventi?" [In linea]. Disponibile: https://www.quora.com/How-does-an-event-loop-work

Leggi Eventloop online: https://riptutorial.com/it/node-js/topic/8652/eventloop

Capitolo 37: Evita l'inferno del callback

Examples

Modulo asincrono

La fonte è disponibile per il download da GitHub. In alternativa, puoi installare usando npm:

$ npm install --save async

Oltre a usare Bower:

$ bower installa async

Esempio:

```
var async = require("async");
async.parallel([
    function(callback) { ... },
    function(callback) { ... }
], function(err, results) {
    // optional callback
});
```

Modulo asincrono

Per fortuna esistono librerie come Async.js per cercare di arginare il problema. Async aggiunge un sottile strato di funzioni in cima al codice, ma può ridurre notevolmente la complessità evitando il nesting di callback.

Esistono molti metodi di supporto in Async che possono essere utilizzati in diverse situazioni, come serie, parallelo, cascata, ecc. Ogni funzione ha un caso d'uso specifico, quindi prenditi un po 'di tempo per imparare quale aiuterà in quali situazioni.

Buono come Async, come qualsiasi cosa, non è perfetto. È molto facile lasciarsi trasportare dalla combinazione di serie, parallelo, per sempre, ecc., A quel punto si è di nuovo a destra dove hai iniziato con il codice disordinato. Fai attenzione a non ottimizzarlo prematuramente. Solo perché alcune attività asincrone possono essere eseguite in parallelo non sempre significa che dovrebbero. In realtà, poiché il nodo è solo a thread singolo, l'esecuzione di attività in parallelo sull'utilizzo di Async ha un guadagno di prestazioni minimo o nullo.

La fonte è disponibile per il download da https://github.com/caolan/async . In alternativa, puoi installare usando npm:

$ npm install --save async

Oltre a usare Bower:

$ bower installa async

Esempio di cascata di Async:

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

var myFile = '/tmp/test';

async.waterfall([
    function(callback) {
        fs.readFile(myFile, 'utf8', callback);
    },
    function(txt, callback) {
        txt = txt + '\nAppended something!';
        fs.writeFile(myFile, txt, callback);
    }
], function (err, result) {
    if(err) return console.log(err);
    console.log('Appended text!');
});
```

Leggi Evita l'inferno del callback online: https://riptutorial.com/it/node-js/topic/10045/evita-l-inferno-del-callback

Capitolo 38: File system I / O

Osservazioni

In Node.js, le operazioni intensive come l'I / O vengono eseguite in *modo asincrono* , ma hanno una controparte *sincrona* (es. Esiste un file `fs.readFile` e la sua controparte è `fs.readFileSync`). Poiché il nodo è a thread singolo, è necessario prestare attenzione quando si utilizzano le operazioni *sincrone* , poiché bloccano l'intero processo.

Se un processo è bloccato da un'operazione sincrona, viene interrotto l'intero ciclo di esecuzione (incluso il ciclo degli eventi). Ciò significa che non verranno eseguiti altri codici asincroni, inclusi eventi e gestori di eventi, e il programma continuerà ad attendere fino al completamento dell'operazione di blocco singolo.

Vi sono usi appropriati sia per le operazioni sincrone che asincrone, ma occorre fare attenzione che vengano utilizzati correttamente.

Examples

Scrivere su un file usando writeFile o writeFileSync

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

// Save the string "Hello world!" in a file called "hello.txt" in
// the directory "/tmp" using the default encoding (utf8).
// This operation will be completed in background and the callback
// will be called when it is either done or failed.
fs.writeFile('/tmp/hello.txt', 'Hello world!', function(err) {
  // If an error occurred, show it and return
  if(err) return console.error(err);
  // Successfully wrote to the file!
});

// Save binary data to a file called "binary.txt" in the current
// directory. Again, the operation will be completed in background.
var buffer = new Buffer([ 0x48, 0x65, 0x6c, 0x6c, 0x6f ]);
fs.writeFile('binary.txt', buffer, function(err) {
  // If an error occurred, show it and return
  if(err) return console.error(err);
  // Successfully wrote binary contents to the file!
});
```

`fs.writeFileSync` comporta in modo simile a `fs.writeFile` , ma non `fs.writeFile` una richiamata poiché completa in modo sincrono e quindi blocca il thread principale. La maggior parte degli sviluppatori di node.js preferiscono le varianti asincrone che non causeranno praticamente alcun ritardo nell'esecuzione del programma.

Nota: il blocco del thread principale è una pratica scorretta in node.js. La funzione sincrona deve essere utilizzata solo durante il debug o quando non sono disponibili altre opzioni.

```
// Write a string to another file and set the file mode to 0755
try {
  fs.writeFileSync('sync.txt', 'anni', { mode: 0o755 });
} catch(err) {
  // An error occurred
  console.error(err);
}
```

Leggi in modo asincrono dai file

Usa il modulo del filesystem per tutte le operazioni sui file:

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

Con codifica

In questo esempio, leggi `hello.txt` dalla directory `/tmp` . Questa operazione sarà completata in background e il callback si verifica al completamento o al fallimento:

```
fs.readFile('/tmp/hello.txt', { encoding: 'utf8' }, (err, content) => {
  // If an error occurred, output it and return
  if(err) return console.error(err);

  // No error occurred, content is a string
  console.log(content);
});
```

Senza codifica

Leggi il file `binary.txt` binario dalla directory corrente, in modo asincrono sullo sfondo. Nota che non impostiamo l'opzione 'encoding' - questo impedisce a Node.js di decodificare il contenuto in una stringa:

```
fs.readFile('binary', (err, binaryContent) => {
  // If an error occurred, output it and return
  if(err) return console.error(err);

  // No error occurred, content is a Buffer, output it in
  // hexadecimal representation.
  console.log(content.toString('hex'));
});
```

Percorsi relativi

Tieni presente che, in generale, lo script può essere eseguito con una directory di lavoro corrente arbitraria. Per indirizzare un file relativo allo script corrente, usa `__dirname` o `__filename` :

```
fs.readFile(path.resolve(__dirname, 'someFile'), (err, binaryContent) => {
```

```
  //Rest of Function
}
```

Elenco dei contenuti della directory con readdir o readdirSync

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

// Read the contents of the directory /usr/local/bin asynchronously.
// The callback will be invoked once the operation has either completed
// or failed.
fs.readdir('/usr/local/bin', (err, files) => {
  // On error, show it and return
  if(err) return console.error(err);

  // files is an array containing the names of all entries
  // in the directory, excluding '.' (the directory itself)
  // and '..' (the parent directory).

  // Display directory entries
  console.log(files.join(' '));
});
```

Una variante sincrona è disponibile come `readdirSync` che blocca il thread principale e quindi impedisce l'esecuzione di codice asincrono allo stesso tempo. La maggior parte degli sviluppatori evita le funzioni di I / O sincrone per migliorare le prestazioni.

```
let files;

try {
  files = fs.readdirSync('/var/tmp');
} catch(err) {
  // An error occurred
  console.error(err);
}
```

Utilizzando un generatore

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

// Iterate through all items obtained via
// 'yield' statements
// A callback is passed to the generator function because it is required by
// the 'readdir' method
function run(gen) {
  var iter = gen((err, data) => {
    if (err) { iter.throw(err); }

    return iter.next(data);
  });

  iter.next();
}

const dirPath = '/usr/local/bin';
```

```
// Execute the generator function
run(function* (resume) {
  // Emit the list of files in the directory from the generator
  var contents = yield fs.readdir(dirPath, resume);
  console.log(contents);
});
```

Lettura da un file in modo sincrono

Per qualsiasi operazione sui file, è necessario il modulo filesystem:

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

Leggere una stringa

`fs.readFileSync` comporta in modo simile a `fs.readFile` , ma non `fs.readFile` una richiamata poiché completa in modo sincrono e quindi blocca il thread principale. La maggior parte degli sviluppatori di node.js preferiscono le varianti asincrone che non causeranno praticamente alcun ritardo nell'esecuzione del programma.

Se viene specificata un'opzione di `encoding` , verrà restituita una stringa, altrimenti verrà restituito un `Buffer` .

```
// Read a string from another file synchronously
let content;
try {
  content = fs.readFileSync('sync.txt', { encoding: 'utf8' });
} catch(err) {
  // An error occurred
  console.error(err);
}
```

Eliminazione di un file utilizzando lo scollegamento o lo scollegamento sincronizzato

Elimina un file in modo asincrono:

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

fs.unlink('/path/to/file.txt', function(err) {
  if (err) throw err;

  console.log('file deleted');
});
```

Puoi anche eliminarlo in modo sincrono *:

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

fs.unlinkSync('/path/to/file.txt');
```

```
console.log('file deleted');
```

* evitare i metodi sincroni perché bloccano l'intero processo fino al termine dell'esecuzione.

Lettura di un file in un buffer utilizzando i flussi

Mentre leggendo il contenuto da un file è già asincrono usando il metodo `fs.readFile()` , a volte vogliamo ottenere i dati in un flusso rispetto a un semplice callback. Questo ci consente di convogliare questi dati in altre posizioni o di elaborarli al loro interno rispetto a tutti in una volta alla fine.

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

// Store file data chunks in this array
let chunks = [];
// We can use this variable to store the final data
let fileBuffer;

// Read file into stream.Readable
let fileStream = fs.createReadStream('text.txt');

// An error occurred with the stream
fileStream.once('error', (err) => {
    // Be sure to handle this properly!
    console.error(err);
});

// File is done being read
fileStream.once('end', () => {
    // create the final data Buffer from data chunks;
    fileBuffer = Buffer.concat(chunks);

    // Of course, you can do anything else you need to here, like emit an event!
});

// Data is flushed from fileStream in chunks,
// this callback will be executed for each chunk
fileStream.on('data', (chunk) => {
    chunks.push(chunk); // push data chunk to array

    // We can perform actions on the partial data we have so far!
});
```

Verifica le autorizzazioni di un file o di una directory

`fs.access()` determina se esiste un percorso e quali autorizzazioni ha un utente per il file o la directory in quel percorso. `fs.access` non restituisce un risultato piuttosto, se non restituisce un errore, il percorso esiste e l'utente ha le autorizzazioni desiderate.

Le modalità di autorizzazione sono disponibili come proprietà sull'oggetto `fs` , `fs.constants`

- `fs.constants.F_OK` - Ha `fs.constants.F_OK` lettura / scrittura / esecuzione (se non viene fornita alcuna modalità, questa è l'impostazione predefinita)
- `fs.constants.R_OK` - Ha permessi di lettura

- `fs.constants.W_OK` - Ha permessi di scrittura
- `fs.constants.X_OK` - Ha permessi di esecuzione (Funziona come `fs.constants.F_OK` su Windows)

in modo asincrono

```
var fs = require('fs');
var path = '/path/to/check';

// checks execute permission
fs.access(path, fs.constants.X_OK, (err) => {
    if (err) {
        console.log("%s doesn't exist", path);
    } else {
        console.log('can execute %s', path);
    }
});
// Check if we have read/write permissions
// When specifying multiple permission modes
// each mode is separated by a pipe : `|`
fs.access(path, fs.constants.R_OK | fs.constants.W_OK, (err) => {
    if (err) {
        console.log("%s doesn't exist", path);
    } else {
        console.log('can read/write %s', path);
    }
});
```

sincrono

`fs.access` ha anche una versione sincrona `fs.accessSync` . Quando si utilizza `fs.accessSync` è necessario racchiuderlo all'interno di un blocco try / catch.

```
// Check write permission
try {
    fs.accessSync(path, fs.constants.W_OK);
    console.log('can write %s', path);
}
catch (err) {
    console.log("%s doesn't exist", path);
}
```

Evitare le condizioni di gara durante la creazione o l'utilizzo di una directory esistente

A causa della natura asincrona del nodo, creando o utilizzando una directory prima:

1. controllando la sua esistenza con `fs.stat()` , quindi
2. creandolo o usandolo in base ai risultati del controllo di esistenza,

può portare a una condizione di competizione se la cartella viene creata tra il momento del controllo e l'ora della creazione. Il metodo riportato di seguito `fs.mkdir()` e `fs.mkdirSync()` in wrapper che catturano gli errori che consentono il passaggio dell'eccezione se il codice è `EEXIST` (già esistente). Se l'errore è qualcos'altro, come `EPERM` (pemission negato), lancia o passa un errore come fanno le funzioni native.

Versione asincrona con `fs.mkdir()`

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

function mkdir (dirPath, callback) {
  fs.mkdir(dirPath, (err) => {
    callback(err && err.code !== 'EEXIST' ? err : null);
  });
}

mkdir('./existingDir', (err) => {

  if (err)
    return console.error(err.code);

  // Do something with `./existingDir` here

});
```

Versione sincrona con `fs.mkdirSync()`

```
function mkdirSync (dirPath) {
  try {
    fs.mkdirSync(dirPath);
  } catch(e) {
    if ( e.code !== 'EEXIST' ) throw e;
  }
}

mkdirSync('./existing-dir');
// Do something with `./existing-dir` now
```

Verifica se esiste un file o una directory

in modo asincrono

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

fs.stat('path/to/file', function(err) {
    if (!err) {
        console.log('file or directory exists');
    }
    else if (err.code === 'ENOENT') {
        console.log('file or directory does not exist');
    }
});
```

sincrono

qui, dobbiamo racchiudere la chiamata di funzione in un blocco `try/catch` per gestire l'errore.

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

try {
    fs.statSync('path/to/file');
    console.log('file or directory exists');
}
catch (err) {
  if (err.code === 'ENOENT') {
    console.log('file or directory does not exist');
  }
}
```

Clonazione di un file utilizzando i flussi

Questo programma illustra come è possibile copiare un file utilizzando flussi leggibili e scrivibili utilizzando le `createReadStream()` e `createWriteStream()` fornite dal modulo del file system.

```
//Require the file System module
var fs = require('fs');

/*
  Create readable stream to file in current directory (__dirname) named 'node.txt'
  Use utf8 encoding
  Read the data in 16-kilobyte chunks
*/
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark:
16 * 1024 });

// create writable stream
var writable = fs.createWriteStream(__dirname + '/nodeCopy.txt');

// Write each chunk of data to the writable stream
readable.on('data', function(chunk) {
    writable.write(chunk);
});
```

Copia di file tramite streaming di piping

Questo programma copia un file utilizzando un flusso leggibile e scrivibile con la funzione `pipe()` fornita dalla classe stream

```
// require the file system module
var fs = require('fs');

/*
    Create readable stream to file in current directory named 'node.txt'
    Use utf8 encoding
    Read the data in 16-kilobyte chunks
*/
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark:
```

```
16 * 1024 });

// create writable stream
var writable = fs.createWriteStream(__dirname + '/nodePipe.txt');

// use pipe to copy readable to writable
readable.pipe(writable);
```

Modifica del contenuto di un file di testo

Esempio. Sostituirà la parola `email` con un `name` in un file di testo `index.txt` con la semplice `replace(/email/gim, 'name')` RegExp `replace(/email/gim, 'name')`

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

fs.readFile('index.txt', 'utf-8', function(err, data) {
    if (err) throw err;

    var newValue = data.replace(/email/gim, 'name');

    fs.writeFile('index.txt', newValue, 'utf-8', function(err, data) {
        if (err) throw err;
        console.log('Done!');
    })
})
```

Determinazione del numero di righe di un file di testo

app.js

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

var file = 'path.to.file';
var linesCount = 0;
var rl = readline.createInterface({
    input: fs.createReadStream(file),
    output: process.stdout,
    terminal: false
});
rl.on('line', function (line) {
    linesCount++; // on each linebreak, add +1 to 'linesCount'
});
rl.on('close', function () {
    console.log(linesCount); // print the result when the 'close' event is called
});
```

Uso:

    nodo app

Leggere un file riga per riga

app.js

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

var file = 'path.to.file';
var rl = readline.createInterface({
    input: fs.createReadStream(file),
    output: process.stdout,
    terminal: false
});

rl.on('line', function (line) {
    console.log(line) // print the content of the line on each linebreak
});
```

Uso:

```
nodo app
```

Leggi File system I / O online: https://riptutorial.com/it/node-js/topic/489/file-system-i---o

Capitolo 39: Gestione degli errori di Node.js

introduzione

Impareremo come creare oggetti Error e come lanciare e gestire gli errori in Node.js

Modifiche future relative alle migliori pratiche nella gestione degli errori.

Examples

Creazione dell'oggetto Error

nuovo errore (messaggio)

Crea un nuovo oggetto errore, in cui il `message` del valore viene impostato sulla proprietà del `message` dell'oggetto creato. Di solito gli argomenti del `message` vengono passati al costruttore Error come stringa. Tuttavia, se l'argomento del `message` è oggetto non una stringa, il costruttore Error chiama il metodo `.toString()` dell'oggetto passato e lo imposta sulla proprietà `message` dell'oggetto error creato.

```
var err = new Error("The error message");
console.log(err.message); //prints: The error message
console.log(err);
//output
//Error: The error message
//    at ...
```

Ogni oggetto di errore ha una traccia di stack. La traccia di stack contiene le informazioni del messaggio di errore e mostra dove si è verificato l'errore (l'output sopra mostra lo stack di errori). Una volta creato l'oggetto errore, il sistema acquisisce la traccia dello stack dell'errore sulla linea corrente. Per ottenere la traccia dello stack, utilizzare la proprietà stack di qualsiasi oggetto di errore creato. Sotto due righe sono identiche:

```
console.log(err);
console.log(err.stack);
```

Errore di lancio

Errore di lancio significa eccezione se nessuna eccezione viene gestita, quindi il server nodo si bloccherà.

La seguente riga genera un errore:

```
throw new Error("Some error occurred");
```

o

```
var err = new Error("Some error occurred");
throw err;
```

o

```
throw "Some error occurred";
```

L'ultimo esempio (tirare le stringhe) non è una buona pratica e non è raccomandato (genera sempre errori che sono istanze di oggetto Error).

Si noti che se si `throw` un errore nel proprio, allora il sistema si bloccherà su quella linea (se non ci sono gestori di eccezioni), nessun codice verrà eseguito dopo quella linea.

```
var a = 5;
var err = new Error("Some error message");
throw err; //this will print the error stack and node server will stop
a++; //this line will never be executed
console.log(a); //and this one also
```

Ma in questo esempio:

```
var a = 5;
var err = new Error("Some error message");
console.log(err); //this will print the error stack
a++;
console.log(a); //this line will be executed and will print 6
```

prova ... cattura blocco

prova ... il blocco catch è per la gestione delle eccezioni, ricorda che l'eccezione indica l'errore generato non l'errore.

```
try {
    var a = 1;
    b++; //this will cause an error because be is undefined
    console.log(b); //this line will not be executed
} catch (error) {
    console.log(error); //here we handle the error caused in the try block
}
```

Nel blocco `try` `b++` causano un errore e quell'errore è passato a `catch` block che può essere gestito lì o addirittura può essere generato lo stesso errore nel catch block o apportare modifiche di poco valore e quindi lanciare. Vediamo il prossimo esempio.

```
try {
    var a = 1;
    b++;
    console.log(b);
} catch (error) {
    error.message = "b variable is undefined, so the undefined can't be incremented"
    throw error;
}
```

Nell'esempio sopra abbiamo modificato la proprietà `message` dell'oggetto `error` e poi lanciato l' `error` modificato.

È possibile attraverso qualsiasi errore nel blocco try e gestirlo nel blocco catch:

```
try {
    var a = 1;
    throw new Error("Some error message");
    console.log(a); //this line will not be executed;
} catch (error) {
    console.log(error); //will be the above thrown error
}
```

Leggi Gestione degli errori di Node.js online: https://riptutorial.com/it/node-js/topic/8590/gestione-degli-errori-di-node-js

Capitolo 40: Gestire la richiesta POST in Node.js

Osservazioni

Node.js utilizza gli stream per gestire i dati in arrivo.

Citando dai documenti,

> Un flusso è un'interfaccia astratta per lavorare con i dati di streaming in Node.js. Il modulo stream fornisce un'API di base che semplifica la creazione di oggetti che implementano l'interfaccia di streaming.

Per gestire il corpo di richiesta di una richiesta POST, utilizzare l'oggetto `request` , che è un flusso leggibile. I flussi di dati vengono emessi come eventi di `data` sull'oggetto `request` .

```
request.on('data', chunk => {
  buffer += chunk;
});
request.on('end', () => {
  // POST request body is now available as `buffer`
});
```

Basta creare una stringa di buffer vuota e aggiungere i dati del buffer come ricevuti tramite eventi di `data` .

NOTA

1. I dati del buffer ricevuti sugli eventi dei `data` sono di tipo Buffer
2. Crea una nuova stringa di buffer per raccogliere dati bufferizzati da eventi di dati **per ogni richiesta,** ovvero crea `buffer` stringa di `buffer` all'interno del gestore di richieste.

Examples

Esempio di server node.js che gestisce solo le richieste POST

```
'use strict';

const http = require('http');

const PORT = 8080;
const server = http.createServer((request, response) => {
  let buffer = '';
  request.on('data', chunk => {
    buffer += chunk;
  });
  request.on('end', () => {
    const responseString = `Received string ${buffer}`;
```

```
    console.log(`Responding with: ${responseString}`);
    response.writeHead(200, "Content-Type: text/plain");
    response.end(responseString);
  });
}).listen(PORT, () => {
  console.log(`Listening on ${PORT}`);
});
```

Leggi Gestire la richiesta POST in Node.js online: https://riptutorial.com/it/node-js/topic/5676/gestire-la-richiesta-post-in-node-js

Capitolo 41: Gestore pacchetti filati

introduzione

Yarn è un gestore di pacchetti per Node.js, simile a npm. Pur condividendo un sacco di terreno comune, ci sono alcune differenze chiave tra Yarn e npm.

Examples

Installazione del filato

Questo esempio spiega i diversi metodi per installare Yarn per il tuo sistema operativo.

Mac OS

homebrew

```
brew update
brew install yarn
```

MacPorts

```
sudo port install yarn
```

Aggiunta di filati al PERCORSO

Aggiungi quanto segue al tuo profilo di shell preferito (`.profile` , `.bashrc` , `.zshrc` etc)

```
export PATH="$PATH:`yarn global bin`"
```

finestre

Installer

Innanzitutto, installa Node.js se non è già installato.

Scarica il programma di installazione di Yarn come `.msi` dal sito Web di Yarn .

cioccolatoso

```
choco install yarn
```

Linux

Debian / Ubuntu

Assicurati che Node.js sia installato per la tua distribuzione, o esegui quanto segue

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Configura il repository YarnPkg

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee
/etc/apt/sources.list.d/yarn.list
```

Installa il filo

```
sudo apt-get update && sudo apt-get install yarn
```

CentOS / Fedora / RHEL

Installa Node.js se non è già installato

```
curl --silent --location https://rpm.nodesource.com/setup_6.x | bash -
```

Installa il filo

```
sudo wget https://dl.yarnpkg.com/rpm/yarn.repo -O /etc/yum.repos.d/yarn.repo
sudo yum install yarn
```

Arco

Installa il filo tramite AUR.

Esempio usando yaourt:

```
yaourt -S yarn
```

Solus

```
sudo eopkg install yarn
```

Tutte le distribuzioni

Aggiungi quanto segue al tuo profilo di shell preferito (`.profile` , `.bashrc` , `.zshrc` etc)

```
export PATH="$PATH:`yarn global bin`"
```

Metodo alternativo di installazione

Script di shell

```
curl -o- -L https://yarnpkg.com/install.sh | bash
```

o specificare una versione da installare

```
curl -o- -L https://yarnpkg.com/install.sh | bash -s -- --version [version]
```

tarball

```
cd /opt
wget https://yarnpkg.com/latest.tar.gz
tar zvxf latest.tar.gz
```

npm

Se hai già installato npm, esegui semplicemente

```
npm install -g yarn
```

Post installazione

Controlla la versione installata di Yarn eseguendo

```
yarn --version
```

Creare un pacchetto base

Il comando `yarn init` ti guiderà attraverso la creazione di un file `package.json` per configurare alcune informazioni sul tuo pacchetto. Questo è simile al comando `npm init` in npm.

Crea e naviga in una nuova directory per conservare il tuo pacchetto, quindi esegui `yarn init`

```
mkdir my-package && cd my-package
yarn init
```

Rispondi alle domande che seguono nella CLI

```
question name (my-package): my-package
question version (1.0.0):
question description: A test package
question entry point (index.js):
question repository url:
question author: StackOverflow Documentation
question license (MIT):
success Saved package.json
  Done in 27.31s.
```

Questo genererà un file `package.json` simile al seguente

```
{
  "name": "my-package",
  "version": "1.0.0",
  "description": "A test package",
  "main": "index.js",
  "author": "StackOverflow Documentation",
  "license": "MIT"
}
```

Ora proviamo ad aggiungere una dipendenza. La sintassi di base per questo è il `yarn add [package-name]`

Eseguire il comando seguente per installare ExpressJS

```
yarn add express
```

Questo aggiungerà una sezione delle `dependencies` al `package.json` e aggiungerà ExpressJS

```
"dependencies": {
    "express": "^4.15.2"
}
```

Installa il pacchetto con Yarn

Il filato usa lo stesso registro usato da npm. Ciò significa che ogni pacchetto disponibile su npm è lo stesso su Yarn.

Per installare un pacchetto, esegui il `yarn add package` .

Se hai bisogno di una versione specifica del pacchetto, puoi utilizzare il `yarn add package@version` .

Se la versione che è necessario installare è stata taggata, è possibile utilizzare il `yarn add package@tag` .

Leggi Gestore pacchetti filati online: https://riptutorial.com/it/node-js/topic/9441/gestore-pacchetti-filati

Capitolo 42: grugnito

Osservazioni

Ulteriori letture:

La guida Installazione grunt contiene informazioni dettagliate sull'installazione di versioni specifiche, di produzione o in fase di sviluppo di Grunt e grunt-cli.

La guida Configurazione delle attività contiene una spiegazione approfondita su come configurare attività, destinazioni, opzioni e file all'interno del Gruntfile, insieme a una spiegazione di modelli, modelli di globbing e importazione di dati esterni.

La guida Creazione delle attività elenca le differenze tra i tipi di attività di Grunt e mostra un numero di attività e configurazioni di esempio.

Examples

Introduzione a GruntJs

Grunt è un Task Runer JavaScript, utilizzato per l'automazione di attività ripetitive come minification, compilation, unit test, linting, ecc.

Per iniziare, ti consigliamo di installare l'interfaccia a riga di comando (CLI) di Grunt a livello globale.

```
npm install -g grunt-cli
```

Preparazione di un nuovo progetto Grunt: una configurazione tipica prevede l'aggiunta di due file al progetto: package.json e Gruntfile.

package.json: questo file viene utilizzato da npm per memorizzare i metadati per i progetti pubblicati come moduli npm. Elencherai i plugin Grunt e Grunt necessari per il tuo progetto come devDependencies in questo file.

Gruntfile: questo file è denominato Gruntfile.js e viene utilizzato per configurare o definire attività e caricare plugin Grunt.

```
Example package.json:

{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-nodeunit": "~0.4.1",
    "grunt-contrib-uglify": "~0.5.0"
```

```
  }
}
```

Esempio gruntfile:

```
module.exports = function(grunt) {

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
      options: {
        banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
      },
      build: {
        src: 'src/<%= pkg.name %>.js',
        dest: 'build/<%= pkg.name %>.min.js'
      }
    }
  });

  // Load the plugin that provides the "uglify" task.
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // Default task(s).
  grunt.registerTask('default', ['uglify']);

};
```

Installazione di gruntplugins

Aggiungere dipendenza

Per usare un gruntplugin, devi prima aggiungerlo come dipendenza al tuo progetto. Usiamo il plugin jshint come esempio.

```
npm install grunt-contrib-jshint --save-dev
```

L'opzione `--save-dev` viene utilizzata per aggiungere il plug-in in `package.json` , in questo modo il plug-in viene sempre installato dopo l' `npm install` .

Caricamento del plugin

Puoi caricare il tuo plugin nel file `loadNpmTasks` usando `loadNpmTasks` .

```
grunt.loadNpmTasks('grunt-contrib-jshint');
```

Configurazione dell'attività

Si configura l'attività nel gruntfile aggiungendo una proprietà chiamata `jshint` all'oggetto passato a `grunt.initConfig` .

```
grunt.initConfig({
```

```
  jshint: {
    all: ['Gruntfile.js', 'lib/**/*.js', 'test/**/*.js']
  }
});
```

Non dimenticare che puoi avere altre proprietà per altri plugin che stai utilizzando.

Esecuzione dell'attività

Per eseguire semplicemente l'attività con il plugin è possibile utilizzare la riga di comando.

```
grunt jshint
```

Oppure puoi aggiungere `jshint` a un'altra attività.

```
grunt.registerTask('default', ['jshint']);
```

L'attività predefinita viene eseguita con il comando grunt nel terminale senza alcuna opzione.

Leggi grugnito online: https://riptutorial.com/it/node-js/topic/6059/grugnito

Capitolo 43: Guida per principianti NodeJS

Examples

Ciao mondo !

Inserire il seguente codice in un nome di file `helloworld.js`

```
console.log("Hello World");
```

Salva il file ed eseguilo tramite Node.js:

```
node helloworld.js
```

Leggi Guida per principianti NodeJS online: https://riptutorial.com/it/node-js/topic/7693/guida-per-principianti-nodejs

Capitolo 44: Hack

Examples

Aggiungi nuove estensioni da richiedere ()

È possibile aggiungere nuove estensioni a `require()` estendendo `require.extensions` .

Per un esempio **XML** :

```
// Add .xml for require()
require.extensions['.xml'] = (module, filename) => {
    const fs = require('fs')
    const xml2js = require('xml2js')

    module.exports = (callback) => {
        // Read required file.
        fs.readFile(filename, 'utf8', (err, data) => {
            if (err) {
                callback(err)
                return
            }
            // Parse it.
            xml2js.parseString(data, (err, result) => {
                callback(null, result)
            })
        })
    }
}
```

Se il contenuto di `hello.xml` è il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
    <bar>baz</bar>
    <qux />
</foo>
```

Puoi leggerlo e analizzarlo tramite `require()` :

```
require('./hello')((err, xml) {
    if (err)
        throw err;
    console.log(err);
})
```

```
{ foo: { bar: [ 'baz' ], qux: [ '' ] } } .
```

Leggi Hack online: https://riptutorial.com/it/node-js/topic/6645/hack

Capitolo 45: http

Examples

server http

Un esempio di base del server HTTP.

scrivi il seguente codice nel file http_server.js:

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

var httpPort = 80;

http.createServer(handler).listen(httpPort, start_callback);

function handler(req, res) {

    var clientIP = req.connection.remoteAddress;
    var connectUsing = req.connection.encrypted ? 'SSL' : 'HTTP';
    console.log('Request received: '+ connectUsing + ' ' + req.method + ' ' + req.url);
    console.log('Client IP: ' + clientIP);

    res.writeHead(200, "OK", {'Content-Type': 'text/plain'});
    res.write("OK");
    res.end();
    return;
}

function start_callback(){
    console.log('Start HTTP on port ' + httpPort)
}
```

quindi dalla tua posizione http_server.js esegui questo comando:

```
node http_server.js
```

dovresti vedere questo risultato:

```
> Start HTTP on port 80
```

ora devi testare il tuo server, devi aprire il tuo browser internet e navigare verso questo URL:

```
http://127.0.0.1:80
```

se la tua macchina esegue un server Linux puoi testarlo in questo modo:

```
curl 127.0.0.1:80
```

dovresti vedere il seguente risultato:

```
ok
```

nella tua console, che esegue l'app, vedrai questi risultati:

```
> Request received: HTTP GET /
> Client IP: ::ffff:127.0.0.1
```

client http

un esempio di base per il client http:

scrivi il codice follwing nel file http_client.js:

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

var options = {
  hostname: '127.0.0.1',
  port: 80,
  path: '/',
  method: 'GET'
};

var req = http.request(options, function(res) {
    console.log('STATUS: ' + res.statusCode);
    console.log('HEADERS: ' + JSON.stringify(res.headers));
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
        console.log('Response: ' + chunk);
    });
    res.on('end', function (chunk) {
        console.log('Response ENDED');
    });
});

req.on('error', function(e) {
    console.log('problem with request: ' + e.message);
});


req.end();
```

quindi dalla tua posizione http_client.js esegui questo comando:

```
node http_client.js
```

dovresti vedere questo risultato:

```
> STATUS: 200
> HEADERS: {"content-type":"text/plain","date":"Thu, 21 Jul 2016 11:27:17
GMT","connection":"close","transfer-encoding":"chunked"}
> Response: OK
> Response ENDED
```

nota: questo esempio dipende dall'esempio del server http.

Leggi http online: https://riptutorial.com/it/node-js/topic/2973/http

Capitolo 46: Iniezione di dipendenza

Examples

Perché utilizzare l'iniezione delle dipendenze

1. **Processo di sviluppo rapido**
2. **Disaccoppiamento**
3. **Scrittura del test unitario**

Processo di sviluppo rapido
Quando si utilizza lo sviluppo di dipendenze, lo sviluppatore del nodo può accelerare i processi di sviluppo perché dopo DI vi è meno conflitto di codice e facile gestione di tutto il modulo.

Disaccoppiamento
I moduli diventano meno coppia, quindi è facile da mantenere.

Scrittura del test unitario
Le dipendenze hardcoded possono passarle nel modulo, quindi è facile scrivere un test unitario per ogni modulo.

Leggi Iniezione di dipendenza online: https://riptutorial.com/it/node-js/topic/7681/iniezione-di-dipendenza

Capitolo 47: Iniziare con la profilatura dei nodi

introduzione

Lo scopo di questo post è iniziare con l'applicazione di profili di profiling e come dare un senso a questi risultati per catturare un bug o una perdita di memoria. Un'applicazione in esecuzione su nodejs non è altro che un processo di motore v8 che è in molti termini simile a un sito Web in esecuzione su un browser e in pratica possiamo acquisire tutte le metriche correlate a un processo di sito Web per un'applicazione di nodo.

Lo strumento di mia preferenza è chrome devtools o chrome inspector accoppiato con il nodo-inspector.

Osservazioni

Il nodo-ispettore non riesce a collegarsi al processo del nodo bebug a volte, nel qual caso non sarà possibile ottenere il punto di interruzione del debug in devtools. Prova ad aggiornare la scheda devtools più volte e attendi qualche secondo per vedere se è in modalità debug.

Se non si riavvia il nodo-inspector dalla riga di comando.

Examples

Creazione di profili di una semplice applicazione di nodo

Passo 1 : Installa il pacchetto node-inspector utilizzando npm globalmente sul tuo computer

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

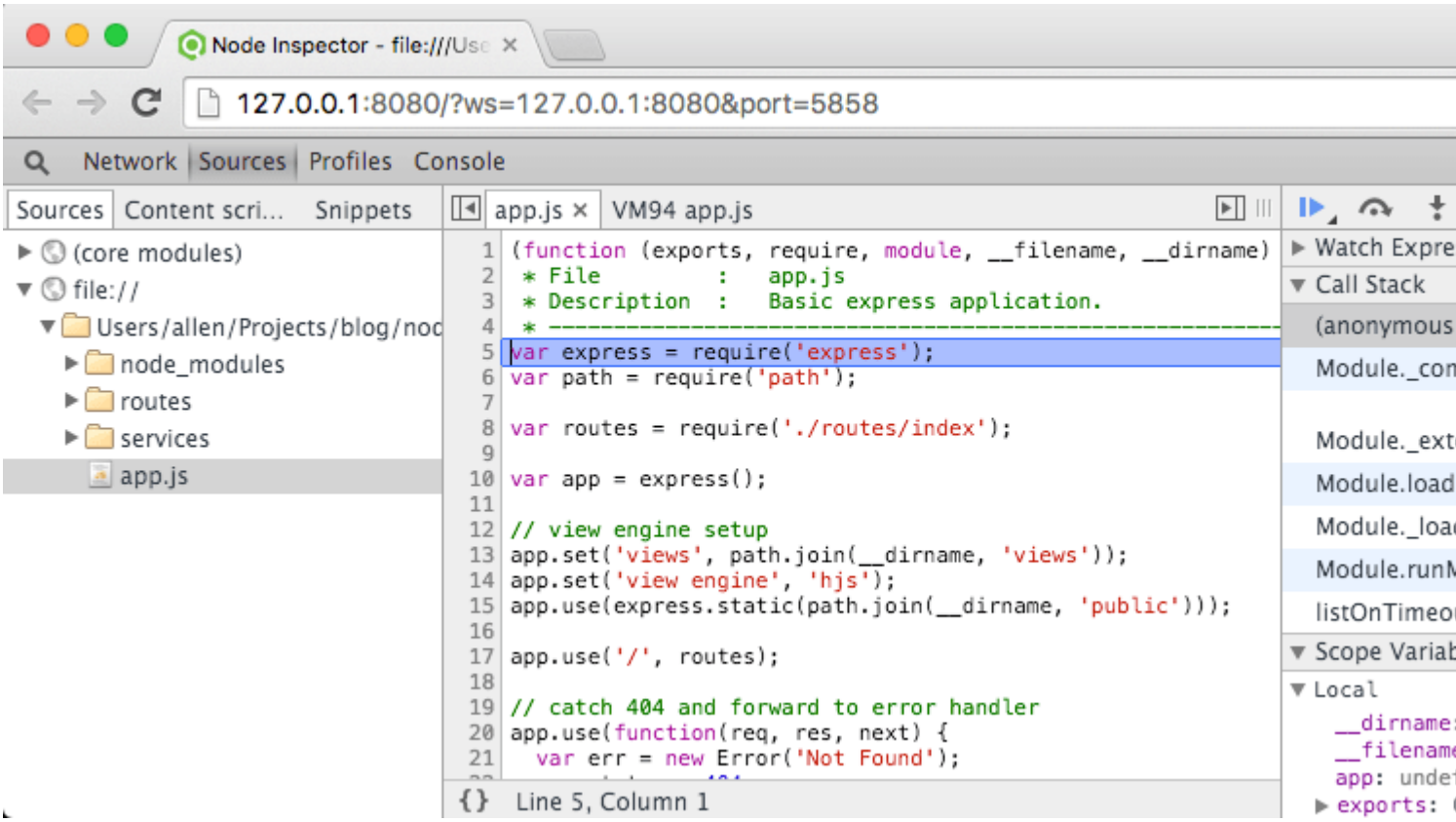
Passaggio 2 : avviare il server ispettore del nodo

```
$ node-inspector
```

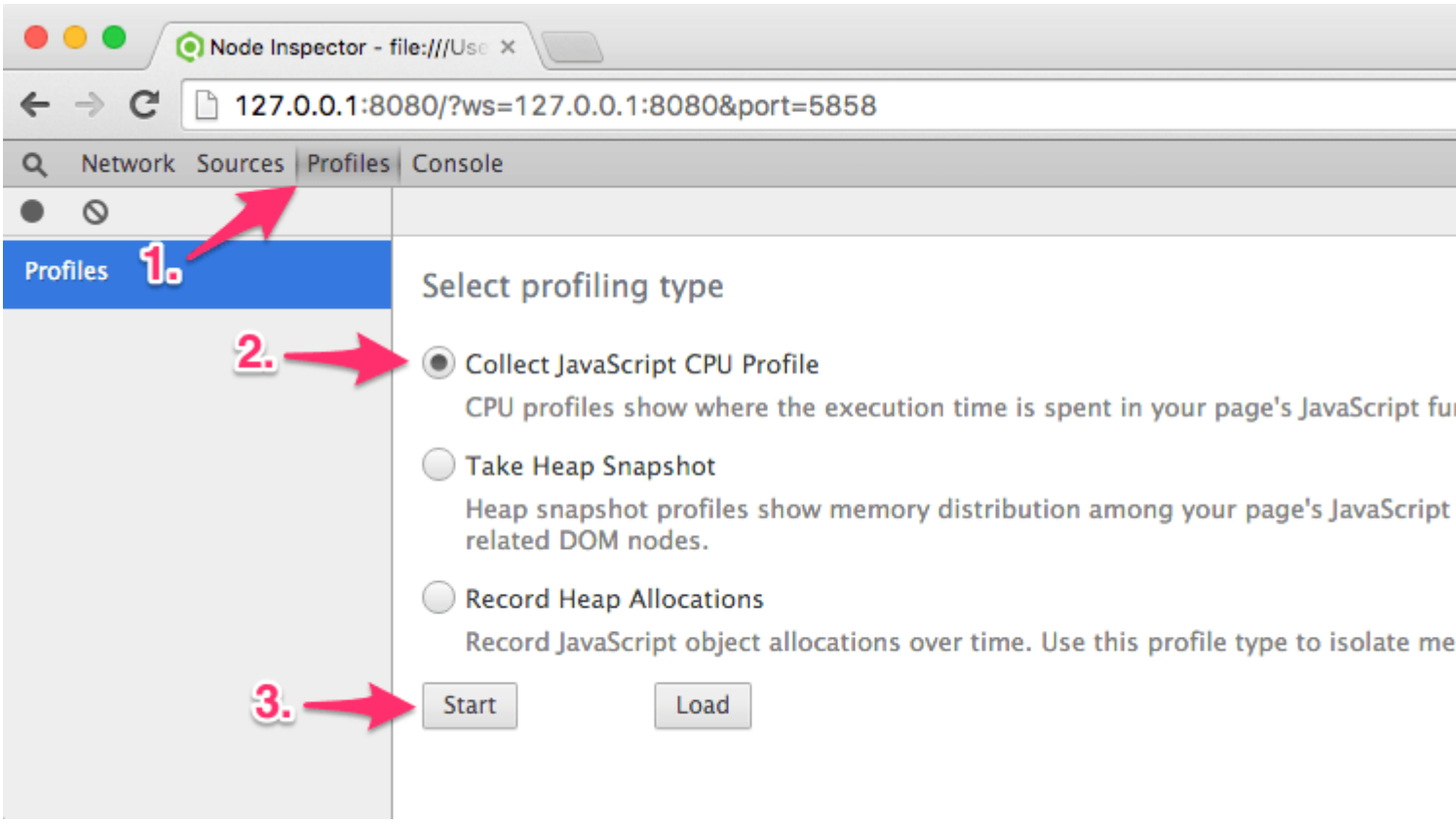
Passaggio 3 : avviare il debug dell'applicazione del nodo

```
$ node --debug-brk your/short/node/script.js
```

Passaggio 4 : apri http://127.0.0.1:8080/?port=5858 nel browser Chrome. E vedrai un'interfaccia di strumenti chrom-dev con il tuo codice sorgente dell'applicazione nodejs nel pannello di sinistra. E poiché abbiamo usato l'opzione debug break durante il debug dell'applicazione, l'esecuzione del codice si fermerà alla prima riga di codice.



Passaggio 5 : questa è la parte facile in cui si passa alla scheda di profilazione e si inizia a profilare l'applicazione. Nel caso in cui si desideri ottenere il profilo per un particolare metodo o flusso, assicurarsi che l'esecuzione del codice sia interrotta poco prima che venga eseguito quel pezzo di codice.



Passaggio 6 : Dopo aver registrato il profilo della CPU o l'heap dump / snapshot o l'allocazione

dell'heap, è possibile visualizzare i risultati nella stessa finestra o salvarli sull'unità locale per un'analisi successiva o un confronto con altri profili.

Puoi usare questi articoli per sapere come leggere i profili:

- Leggere i profili della CPU
- Profiler per CPU di Chrome e profilatore di heap

Leggi Iniziare con la profilatura dei nodi online: https://riptutorial.com/it/node-js/topic/9347/iniziare-con-la-profilatura-dei-nodi

Capitolo 48: Installare Node.js

Examples

Installa Node.js su Ubuntu

Utilizzando il gestore di pacchetti apt

```
sudo apt-get update
sudo apt-get install nodejs
sudo apt-get install npm
sudo ln -s /usr/bin/nodejs /usr/bin/node

# the node & npm versions in apt are outdated. This is how you can update them:
sudo npm install -g npm
sudo npm install -g n
sudo n stable # (or lts, or a specific version)
```

Utilizzando l'ultima versione specifica (ad es. LTS 6.x) direttamente da nodesource

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
apt-get install -y nodejs
```

Inoltre, per il modo corretto di installare i moduli globali di npm, impostare la directory personale per loro (elimina la necessità di sudo ed evita gli errori EACCES):

```
mkdir ~/.npm-global
echo "export PATH=~/.npm-global/bin:$PATH" >> ~/.profile
source ~/.profile
npm config set prefix '~/.npm-global'
```

Installazione di Node.js su Windows

Installazione standard

Tutti i binari, gli installatori e i file sorgente di Node.js possono essere scaricati qui .

È possibile scaricare solo il runtime `node.exe` o utilizzare il programma di installazione di Windows (`.msi`), che installerà anche `npm` , il gestore pacchetti consigliato per Node.js e i percorsi di configurazione.

Installazione tramite gestore pacchetti

È inoltre possibile installare dal gestore pacchetti *Chocolatey* (Software Management Automation).

```
# choco install nodejs.install
```

Maggiori informazioni sulla versione attuale, puoi trovare nel repository choco qui .

Utilizzo di Node Version Manager (nvm)

Node Version Manager , altrimenti noto come nvm, è uno script di bash che semplifica la gestione di più versioni di Node.js.

Per installare nvm, utilizzare lo script di installazione fornito:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Per Windows esiste un pacchetto nvm-windows con un programma di installazione. Questa pagina GithHub contiene i dettagli per l'installazione e l'utilizzo del pacchetto nvm-windows.

Dopo aver installato nvm, esegui "nvm on" dalla riga di comando. Ciò consente a nvm di controllare le versioni del nodo.

Nota: potrebbe essere necessario riavviare il terminale affinché riconosca il comando `nvm` appena installato.

Quindi installare la versione più recente del nodo:

```
$ nvm install node
```

Puoi anche installare una versione specifica del nodo, passando le versioni major, minor e / o patch:

```
$ nvm install 6
$ nvm install 4.2
```

Per elencare le versioni disponibili per l'installazione:

```
$ nvm ls-remote
```

È quindi possibile passare versioni passando la versione nello stesso modo in cui si esegue l'installazione:

```
$ nvm use 5
```

Puoi impostare una versione specifica del nodo che hai installato come **versione predefinita** inserendo:

```
$ nvm alias default 4.2
```

Per visualizzare un elenco di versioni di nodo installate sul computer, immettere:

```
$ nvm ls
```

Per utilizzare versioni di nodi specifiche del progetto, è possibile salvare la versione nel file .nvmrc. In questo modo, iniziare a lavorare con un altro progetto sarà meno soggetto ad errori dopo averlo scaricato dal suo repository.

```
$ echo "4.2" > .nvmrc
$ nvm use
Found '/path/to/project/.nvmrc' with version <4.2>
Now using node v4.2 (npm v3.7.3)
```

Quando Node viene installato tramite nvm, non è necessario utilizzare `sudo` per installare i pacchetti globali poiché sono installati nella cartella principale. Quindi `npm i -g http-server` funziona senza errori di autorizzazione.

Installa Node.js da Source con il gestore pacchetti APT

Prerequisiti

```
sudo apt-get install build-essential
sudo apt-get install python

[optional]
sudo apt-get install git
```

Ottieni la fonte e costruisci

```
cd ~
git clone https://github.com/nodejs/node.git
```

OPPURE Per l'ultima versione di Node.js LTS 6.10.2

```
cd ~
wget https://nodejs.org/dist/v6.3.0/node-v6.10.2.tar.gz
tar -xzvf node-v6.10.2.tar.gz
```

Passare alla directory di origine come in `cd ~/node-v6.10.2`

```
./configure
make
sudo make install
```

Installare Node.js su Mac usando il gestore pacchetti

homebrew

Puoi installare Node.js utilizzando il gestore di pacchetti Homebrew .

Inizia aggiornando brew:

```
brew update
```

Potrebbe essere necessario modificare autorizzazioni o percorsi. È meglio eseguirlo prima di procedere:

```
brew doctor
```

Successivamente è possibile installare Node.js eseguendo:

```
brew install node
```

Una volta installato Node.js, è possibile convalidare la versione installata eseguendo:

```
node -v
```

macports

Puoi anche installare node.js tramite Macports .

Innanzitutto aggiornalo per assicurarti che i pacchetti più recenti siano referenziati:

```
sudo port selfupdate
```

Quindi installare nodejs e npm

```
sudo port install nodejs npm
```

È ora possibile eseguire il nodo tramite CLI direttamente richiamando il `node` . Inoltre, puoi controllare la versione corrente del tuo nodo con

```
node -v
```

Installazione tramite MacOS X Installer

Puoi trovare i programmi di installazione nella pagina di download di Node.js. Normalmente, Node.js raccomanda due versioni di Node, la versione LTS (supporto a lungo termine) e la versione corrente (ultima versione). Se si è nuovi a Node, basta andare su LTS e quindi fare clic sul pulsante `Macintosh Installer` per scaricare il pacchetto.

Se vuoi trovare altre versioni di NodeJS, vai qui , scegli la tua versione, quindi fai clic su Scarica. Dalla pagina di download, cerca un file con estensione `.pkg` .

Una volta scaricato il file (con estensione `.pkg` ofcourse), fai doppio clic per installarlo. Il programma di installazione compresso con `Node.js` e `npm` , per impostazione predefinita, il pacchetto verrà installato entrambi, ma è possibile personalizzare quale installare facendo clic sul pulsante `customize` nel passaggio `Installation Type` . Oltre a questo, basta seguire le istruzioni di installazione, è piuttosto semplice.

Controlla se il Nodo è installato

`terminal` aperto (se non sai come aprire il tuo terminale, guarda questo wikihow). Nel `node --version` tipo terminale `node --version` quindi immettere. Il tuo terminale apparirà così se Node è installato:

```
$ node --version
v7.2.1
```

La versione `v7.2.1` è la tua Node.js, se ricevi il `command not found: node` message `command not found: node` invece di quello, allora significa che c'è un problema con la tua installazione.

Installazione di Node.js su Raspberry PI

Per installare v6.x aggiornare i pacchetti

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
```

Utilizzando il gestore di pacchetti apt

```
sudo apt-get install -y nodejs
```

Installazione con Node Version Manager in Fish Shell con Oh My Fish!

Node Version Manager (`nvm`) semplifica enormemente la gestione delle versioni di Node.js, la loro installazione e rimuove la necessità di `sudo` quando si ha a che fare con pacchetti (ad es. `npm install ...`). Fish Shell (`fish`) " *è una shell da riga di comando intelligente e user-friendly per OS X, Linux e il resto della famiglia* " che è un'alternativa popolare tra i programmatori alle shell comuni come `bash` . Infine, Oh My Fish (`omf`) consente di personalizzare e installare pacchetti all'interno di Fish Shell.

Questa guida presuppone che tu stia già utilizzando Fish come shell .

Installa nvm

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash
```

Installa Oh My Fish

```
curl -L https://github.com/oh-my-fish/oh-my-fish/raw/master/bin/install | fish
```

(Nota: a questo punto ti verrà richiesto di riavviare il tuo terminale. Vai avanti e fallo ora.)

Installa plugin-nvm per Oh My Fish

Installeremo plugin-nvm tramite Oh My Fish per esporre `nvm` funzionalità di `nvm` all'interno di Fish Shell:

```
omf install nvm
```

Installa Node.js con Node Version Manager

Ora sei pronto per usare `nvm` . Puoi installare e utilizzare la versione di Node.js di tuo gradimento. Qualche esempio:

- Installa la versione più recente del nodo: `nvm install node`
- Installare specificamente la versione 6.3.1: `nvm install 6.3.1`
- Elenca le verten installate: `nvm ls`
- Passare a 4.3.1: `nvm use 4.3.1` installato in precedenza `nvm use 4.3.1`

Note finali

Ricorda ancora che non abbiamo più bisogno di `sudo` quando si ha a che fare con Node.js usando questo metodo! Le versioni del nodo, i pacchetti e così via sono installati nella tua home directory.

Installa Node.js dal sorgente su Centos, RHEL e Fedora

Prerequisiti

- idiota
- `clang` and `clang++` 3.4 ^ o `gcc` e `g++` 4.8 ^
- Python 2.6 o 2.7
- GNU Make 3.81 ^

Ottieni la fonte

Node.js v6.x LTS

```
git clone -b v6.x https://github.com/nodejs/node.git
```

Node.js v7.x

```
git clone -b v7.x https://github.com/nodejs/node.git
```

Costruire

```
cd node
./configure
make -jX
su -c make install
```

X: il numero di core del processore, velocizza notevolmente la build

Pulizia [Opzionale]

```
cd
rm -rf node
```

Installare Node.js con n

Innanzitutto, c'è un involucro davvero bello per l'installazione di `n` sul tuo sistema. Corri:

```
curl -L https://git.io/n-install | bash
```

installare `n` . Quindi installa i binari in vari modi:

più recente

```
n latest
```

stabile

```
n stable
```

è

```
n lts
```

Qualsiasi altra versione

```
n <version>
```

per esempio `n 4.4.7`

Se questa versione è già installata, questo comando attiverà quella versione.

Versioni di commutazione

`n` da solo produrrà un elenco di selezione di binari installati. Usa su e giù per trovare quello che vuoi e Enter per attivarlo.

Leggi Installare Node.js online: https://riptutorial.com/it/node-js/topic/1294/installare-node-js

Capitolo 49: Instradare richieste ajax con Express.JS

Examples

Una semplice implementazione di AJAX

Dovresti avere il modello generatore di base espresso

In app.js, aggiungi (puoi aggiungerlo ovunque dopo `var app = express.app()`):

```
app.post(function(req, res, next){
    next();
});
```

Ora nel tuo file index.js (o nella sua rispettiva corrispondenza), aggiungi:

```
router.get('/ajax', function(req, res){
    res.render('ajax', {title: 'An Ajax Example', quote: "AJAX is great!"});
});
router.post('/ajax', function(req, res){
    res.render('ajax', {title: 'An Ajax Example', quote: req.body.quote});
});
```

Creare un file `ajax.jade` / `ajax.pug` o `ajax.ejs` nella directory `/views` , aggiungere:

Per Jade / PugJS:

```
extends layout
script(src="http://code.jquery.com/jquery-3.1.0.min.js")
script(src="/magic.js")
h1 Quote: !{quote}
form(method="post" id="changeQuote")
    input(type='text', placeholder='Set quote of the day', name='quote')
    input(type="submit", value="Save")
```

Per EJS:

```
<script src="http://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="/magic.js"></script>
<h1>Quote: <%=quote%> </h1>
<form method="post" id="changeQuote">
    <input type="text" placeholder="Set quote of the day" name="quote"/>
    <input type="submit" value="Save">
</form>
```

Ora crea un file in `/public` chiamato `magic.js`

```
$(document).ready(function(){
```

```
    $("form#changeQuote").on('submit', function(e){
        e.preventDefault();
        var data = $('input[name=quote]').val();
        $.ajax({
            type: 'post',
            url: '/ajax',
            data: data,
            dataType: 'text'
        })
        .done(function(data){
            $('h1').html(data.quote);
        });
    });
});
```

E il gioco è fatto! Quando fai clic su Salva, il preventivo cambierà!

Leggi Instradare richieste ajax con Express.JS online: https://riptutorial.com/it/node-js/topic/6738/instradare-richieste-ajax-con-express-js

Capitolo 50: Integrazione con Mongodb

Sintassi

- db. *collezione* .insertOne (*documento* , *opzioni (w, wtimeout, j, serializeFuntions, forceServerObjectId, bypassDocumentValidation)* , *callback*)
- db. *collection* .insertMany (*[documenti]* , *opzioni (w, wtimeout, j, serializeFuntions, forceServerObjectId, bypassDocumentValidation)* , *callback*)
- db. *collezione* .find (*query*)
- db. *collection* .updateOne (*filtro* , *aggiornamento* , *opzioni (upsert, w, wtimeout, j, bypassDocumentValidation)* , *callback*)
- db. *collection.updateMany* (*filtro* , *aggiornamento* , *opzioni (upsert, w, wtimeout, j)* , *callback*)
- db. *collezione* .deleteOne (*filtro* , *opzioni (upsert, w, wtimeout, j)* , *callback*)
- db. *collection* .deleteMany (*filtro* , *opzioni (upsert, w, wtimeout, j)* , *callback*)

Parametri

Parametro	Dettagli
documento	Un oggetto javascript che rappresenta un documento
documenti	Una serie di documenti
domanda	Un oggetto che definisce una query di ricerca
filtro	Un oggetto che definisce una query di ricerca
richiama	Funzione da chiamare quando l'operazione è terminata
opzioni	*(facoltativo)* Impostazioni opzionali *(predefinito: null)*
w	*(facoltativo)* La preoccupazione di scrittura
wtimeout	*(facoltativo)* Timeout di scrittura. *(valore predefinito: null)*
j	*(facoltativo)* Specificare un problema di scrittura del diario *(predefinito: falso)*
upsert	*(facoltativo)* Operazione di aggiornamento *(impostazione predefinita: falso)*
Multi	*(facoltativo)* Aggiorna uno / tutti i documenti *(predefinito: falso)*
serializeFunctions	*(opzionale)* Serializza le funzioni su qualsiasi oggetto *(predefinito: falso)*

Parametro	Dettagli
forceServerObjectId	*(facoltativo)* Forza server per assegnare valori _id anziché driver *(predefinito: falso)*
bypassDocumentValidation	*(facoltativo)* Permetti al driver di bypassare la convalida dello schema in MongoDB 3.2 o successiva *(predefinito: falso)*

Examples

Connetti a MongoDB

Connettiti a MongoDB, stampa 'Connesso!' e chiudi la connessione.

```
const MongoClient = require('mongodb').MongoClient;

var url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function(err, db) { // MongoClient method 'connect'
    if (err) throw new Error(err);
    console.log("Connected!");
    db.close(); // Don't forget to close the connection when you are done
});
```

Metodo MongoClient `Connect()`

MongoClient.connect (*url* , *opzioni* , *callback*)

Discussione	genere	Descrizione
`url`	stringa	Una stringa che specifica l'ip / hostname del server, la porta e il database
`options`	oggetto	*(facoltativo)* Impostazioni opzionali *(predefinito: null)*
`callback`	Funzione	Funzione da chiamare quando viene effettuato il tentativo di connessione

La funzione di `callback` accetta due argomenti

- `err` : Errore - Se si verifica un `err` , verrà definito l'argomento `err`
- `db` : object - L'istanza di MongoDB

Inserisci un documento

Inserisci un documento chiamato "myFirstDocument" e imposta **2** proprietà, `greetings` e `farewell`

```
const MongoClient = require('mongodb').MongoClient;
```

```
const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').insertOne({ // Insert method 'insertOne'
    "myFirstDocument": {
      "greetings": "Hellu",
      "farewell": "Bye"
    }
  }, function (err, result) {
    if (err) throw new Error(err);
    console.log("Inserted a document into the myCollection collection!");
    db.close(); // Don't forget to close the connection when you are done
  });
});
```

Metodo di raccolta `insertOne()`

db.collection (*collection*) .insertOne (*documento* , *opzioni* , *callback*)

Discussione	genere	Descrizione
`collection`	stringa	Una stringa che specifica la collezione
`document`	oggetto	Il documento da inserire nella collezione
`options`	oggetto	*(facoltativo)* Impostazioni opzionali *(predefinito: null)*
`callback`	Funzione	Funzione da chiamare quando l'operazione di inserimento è terminata

La funzione di `callback` accetta due argomenti

- `err` : Errore - Se si verifica un `err` , verrà definito l'argomento `err`
- `result` : oggetto - Un oggetto contenente dettagli sull'operazione di inserimento

Leggi una collezione

Ottieni tutti i documenti nella raccolta "myCollection" e stampali sulla console.

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  var cursor = db.collection('myCollection').find(); // Read method 'find'
  cursor.each(function (err, doc) {
    if (err) throw new Error(err);
    if (doc != null) {
      console.log(doc); // Print all documents
    } else {
```

```
      db.close(); // Don't forget to close the connection when you are done
    }
  });
});
```

Metodo di raccolta `find()`

db.collection (*collection*) .find ()

Discussione	genere	Descrizione
`collection`	stringa	Una stringa che specifica la collezione

Aggiorna un documento

Trova un documento con la proprietà `{ greetings: 'Hellu' }` e `{ greetings: 'Hellu' }` in `{ greetings: 'Whut?' }`

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
    db.collection('myCollection').updateOne({ // Update method 'updateOne'
        greetings: "Hellu" },
        { $set: { greetings: "Whut?" }},
        function (err, result) {
            if (err) throw new Error(err);
            db.close(); // Don't forget to close the connection when you are done
        });
});
```

Metodo di raccolta `updateOne()`

db.collection (*collection*) .updateOne (*filtro* , *aggiornamento* , *opzioni callback*)

Parametro	genere	Descrizione
`filter`	oggetto	Specifica i criteri di selezione
`update`	oggetto	Specifica le modifiche da applicare
`options`	oggetto	*(facoltativo)* Impostazioni opzionali *(predefinito: null)*
`callback`	Funzione	Funzione da chiamare quando l'operazione è terminata

La funzione di `callback` accetta due argomenti

- `err` : Errore - Se si verifica un errore, verrà definito l'argomento err
- `db` : object - L'istanza di MongoDB

Elimina un documento

Elimina un documento con la proprietà `{ greetings: 'Whut?' }`

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
    db.collection('myCollection').deleteOne(// Delete method 'deleteOne'
        { greetings: "Whut?" },
        function (err, result) {
            if (err) throw new Error(err);
            db.close(); // Don't forget to close the connection when you are done
    });
});
```

Metodo di raccolta `deleteOne()`

db.collection (*collection*) .deleteOne (*filtro* , *opzioni* , *callback*)

Parametro	genere	Descrizione
`filter`	oggetto	Un documento che specifica i criteri di selezione
`options`	oggetto	*(facoltativo)* Impostazioni opzionali *(predefinito: null)*
`callback`	Funzione	Funzione da chiamare quando l'operazione è terminata

La funzione di `callback` accetta due argomenti

- `err` : Errore - Se si verifica un errore, verrà definito l'argomento err
- `db` : object - L'istanza di MongoDB

Elimina più documenti

Elimina TUTTI i documenti con una proprietà 'addio' impostata su 'ok'.

```
const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
    db.collection('myCollection').deleteMany(// MongoDB delete method 'deleteMany'
        { farewell: "okay" }, // Delete ALL documents with the property 'farewell: okay'
        function (err, result) {
```

```
            if (err) throw new Error(err);
            db.close(); // Don't forget to close the connection when you are done
    });
});
```

Metodo di raccolta `deleteMany()`

db.collection (*collection*) .deleteMany (*filtro* , *opzioni* , *callback*)

Parametro	genere	Descrizione
`filter`	documento	Un documento che specifica i criteri di selezione
`options`	oggetto	*(facoltativo)* Impostazioni opzionali *(predefinito: null)*
`callback`	funzione	Funzione da chiamare quando l'operazione è terminata

La funzione di `callback` accetta due argomenti

- `err` : Errore - Se si verifica un errore, verrà definito l'argomento err
- `db` : object - L'istanza di MongoDB

Semplice connessione

```
MongoDB.connect('mongodb://localhost:27017/databaseName', function(error, database) {
   if(error) return console.log(error);
   const collection = database.collection('collectionName');
   collection.insert({key: 'value'}, function(error, result) {
      console.log(error, result);
   });
});
```

Connessione semplice, usando le promesse

```
const MongoDB = require('mongodb');

MongoDB.connect('mongodb://localhost:27017/databaseName')
    .then(function(database) {
        const collection = database.collection('collectionName');
        return collection.insert({key: 'value'});
    })
    .then(function(result) {
        console.log(result);
    });
    ```
```

Leggi Integrazione con Mongodb online: https://riptutorial.com/it/node-js/topic/5002/integrazione-con-mongodb
```

Capitolo 51: Integrazione con MySQL

introduzione

In questo argomento imparerai come integrarsi con Node.js utilizzando lo strumento di gestione del database MYSQL. Imparerai vari modi per connetterti e interagire con i dati che risiedono in mysql usando un programma e uno script nodejs.

Examples

Interrogare un oggetto di connessione con parametri

Quando si desidera utilizzare il contenuto generato dall'utente in SQL, con i parametri. Ad esempio per cercare un utente con il nome `aminadav` dovresti fare:

```
var username = 'aminadav';
var querystring = 'SELECT name, email from users where name = ?';
connection.query(querystring, [username], function(err, rows, fields) {
  if (err) throw err;
  if (rows.length) {
    rows.forEach(function(row) {
      console.log(row.name, 'email address is', row.email);
    });
  } else {
    console.log('There were no results.');
  }
});
```

Utilizzo di un pool di connessioni

un. Esecuzione di più query contemporaneamente

Tutte le query nella connessione MySQL vengono eseguite una dopo l'altra. Significa che se si desidera eseguire 10 query e ciascuna query richiede 2 secondi, saranno necessari 20 secondi per completare l'intera esecuzione. La soluzione è creare 10 connessioni ed eseguire ogni query in una connessione diversa. Questo può essere fatto automaticamente usando il pool di connessioni

```
var pool  = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass',
  database        : 'schema'
});

for(var i=0;i<10;i++){
  pool.query('SELECT ` as example', function(err, rows, fields) {
    if (err) throw err;
```

```
    console.log(rows[0].example); //Show 1
  });
 }
```

Eseguirà tutte le 10 query in parallelo.

Quando usi `pool` non hai più bisogno della connessione. Puoi interrogare direttamente il pool. Il modulo MySQL cercherà la prossima connessione gratuita per eseguire la query.

b. Raggiungimento della multi-tenancy sul server di database con diversi database ospitati su di esso.

La multitenancy è un requisito comune delle applicazioni aziendali al giorno d'oggi e la creazione di un pool di connessioni per ogni database nel server di database non è consigliata. quindi, ciò che possiamo fare è creare un pool di connessioni con un server di database e quindi cambiarle tra i database ospitati su un server di database su richiesta.

Supponiamo che la nostra applicazione abbia diversi database per ogni azienda ospitata sul server di database. Ci collegheremo al rispettivo database aziendale quando l'utente raggiunge l'applicazione. Ecco l'esempio su come farlo: -

```
var pool  = mysql.createPool({
      connectionLimit : 10,
      host            : 'example.org',
      user            : 'bobby',
      password        : 'pass'
    });

pool.getConnection(function(err, connection){
    if(err){
        return cb(err);
    }
    connection.changeUser({database : "firm1"});
    connection.query("SELECT * from history", function(err, data){
        connection.release();
        cb(err, data);
    });
});
```

Lasciatemi fare un esempio:

Durante la definizione della configurazione del pool non ho fornito il nome del database, ma ho fornito solo il server del database

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

quindi, quando vogliamo utilizzare il database specifico sul server di database, chiediamo la

connessione al database dei risultati utilizzando: -

```
    connection.changeUser({database : "firm1"});
```

puoi consultare la documentazione ufficiale qui

Connetti a MySQL

Uno dei modi più semplici per connettersi a MySQL è usando il modulo `mysql` . Questo modulo gestisce la connessione tra l'app Node.js e il server MySQL. Puoi installarlo come qualsiasi altro modulo:

```
npm install --save mysql
```

Ora devi creare una connessione mysql, che puoi in seguito interrogare.

```
const mysql      = require('mysql');
const connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'me',
  password : 'secret',
  database : 'database_schema'
});

connection.connect();

// Execute some query statements
// I.e. SELECT * FROM FOO

connection.end();
```

Nel prossimo esempio imparerai come interrogare l'oggetto di `connection` .

Interrogare un oggetto di connessione senza parametri

Si invia la query come stringa e in risposta si riceve una risposta con la risposta. Il callback ti dà `error` , array di `rows` e campi. Ogni riga contiene tutta la colonna della tabella restituita. Ecco uno snippet per la seguente spiegazione.

```
connection.query('SELECT name,email from users', function(err, rows, fields) {
  if (err) throw err;

  console.log('There are:', rows.length,' users');
  console.log('First user name is:',rows[0].name)
});
```

Esegui un numero di query con una singola connessione da un pool

Potrebbero esserci situazioni in cui hai impostato un pool di connessioni MySQL, ma hai un numero di query che vorresti eseguire in sequenza:

```
SELECT 1;
SELECT 2;
```

Si *potrebbe* semplicemente eseguire quindi usando `pool.query` come visto altrove , tuttavia se si dispone di una sola connessione libera nel pool, è necessario attendere fino a quando una connessione diventa disponibile prima di poter eseguire la seconda query.

Tuttavia, è possibile mantenere una connessione attiva dal pool ed eseguire tutte le query che si desidera utilizzando una singola connessione mediante `pool.getConnection` :

```
pool.getConnection(function (err, conn) {
  if (err) return callback(err);

  conn.query('SELECT 1 AS seq', function (err, rows) {
    if (err) throw err;

    conn.query('SELECT 2 AS seq', function (err, rows) {
      if (err) throw err;

      conn.release();
      callback();
    });
  });
});
```

Nota: è necessario ricordare di `release` la connessione, altrimenti è disponibile una connessione MySQL in meno per il resto del pool!

Per ulteriori informazioni sul pooling delle connessioni MySQL, consultare i documenti MySQL .

Restituisce la query quando si verifica un errore

È possibile allegare la query eseguita all'oggetto `err` quando si verifica un errore:

```
var q = mysql.query('SELECT `name` FROM `pokedex` WHERE `id` = ?', [ 25 ], function (err,
result) {
  if (err) {
    // Table 'test.pokedex' doesn't exist
    err.query = q.sql; // SELECT `name` FROM `pokedex` WHERE `id` = 25
    callback(err);
  }
  else {
    callback(null, result);
  }
});
```

Esporta pool di connessioni

```
// db.js

const mysql = require('mysql');

const pool = mysql.createPool({
```

```
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bob',
  password        : 'secret',
  database        : 'my_db'
});

module.export = {
  getConnection: (callback) => {
    return pool.getConnection(callback);
  }
}
```

```
// app.js

const db = require('./db');

db.getConnection((err, conn) => {
  conn.query('SELECT something from sometable', (error, results, fields) => {
    // get the results
    conn.release();
  });
});
```

Leggi Integrazione con MySQL online: https://riptutorial.com/it/node-js/topic/1406/integrazione-con-mysql

Capitolo 52: Integrazione del passaporto

Osservazioni

La password deve essere **sempre** sottoposta a hash. Un modo semplice per proteggere le password usando **NodeJS** sarebbe utilizzare il modulo **bcrypt-nodejs** .

Examples

Iniziare

Il passaporto deve essere inizializzato usando il middleware `passport.initialize()` . Per utilizzare le sessioni di accesso, è necessario il middleware `passport.session()` .

Si noti che devono essere definiti i metodi `passport.serialize()` e `passport.deserializeUser()` . **Passport** serializzerà e deserializzerà le istanze utente da e verso la sessione

```
const express = require('express');
const session = require('express-session');
const passport = require('passport');
const cookieParser = require('cookie-parser');
const app = express();

// Required to read cookies
app.use(cookieParser());

passport.serializeUser(function(user, next) {
    // Serialize the user in the session
    next(null, user);
});

passport.deserializeUser(function(user, next) {
    // Use the previously serialized user
        next(null, user);
});

// Configuring express-session middleware
app.use(session({
    secret: 'The cake is a lie',
    resave: true,
    saveUninitialized: true
}));

// Initializing passport
app.use(passport.initialize());
app.use(passport.session());

// Starting express server on port 3000
app.listen(3000);
```

Autenticazione locale

Il modulo **passport-local** viene utilizzato per implementare un'autenticazione locale.

> Questo modulo ti consente di autenticarti usando un nome utente e una password nelle tue applicazioni Node.js.

Registrazione dell'utente:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

// A named strategy is used since two local strategy are used :
// one for the registration and the other to sign-in
passport.use('localSignup', new LocalStrategy({
    // Overriding defaults expected parameters,
    // which are 'username' and 'password'
    usernameField: 'email',
    passwordField: 'password',
    passReqToCallback: true // allows us to pass back the entire request to the callback    .
},
function(req, email, password, next) {
    // Check in database if user is already registered
    findUserByEmail(email, function(user) {
        // If email already exists, abort registration process and
        // pass 'false' to the callback
        if (user) return next(null, false);
        // Else, we create the user
        else {
            // Password must be hashed !
            let newUser = createUser(email, password);

            newUser.save(function() {
                // Pass the user to the callback
                return next(null, newUser);
            });
        }
    });
});
```

Accesso dell'utente:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

passport.use('localSignin', new LocalStrategy({
    usernameField : 'email',
    passwordField : 'password',
},
function(email, password, next) {
    // Find the user
    findUserByEmail(email, function(user) {
        // If user is not found, abort signing in process
        // Custom messages can be provided in the verify callback
        // to give the user more details concerning the failed authentication
        if (!user)
            return next(null, false, {message: 'This e-mail address is not associated with any
account.'});
        // Else, we check if password is valid
        else {
```

```
            // If password is not correct, abort signing in process
            if (!isPasswordValid(password)) return next(null, false);
            // Else, pass the user to callback
            else return next(null, user);
        }
    });
});
```

Creare percorsi:

```
// ...
app.use(passport.initialize());
app.use(passport.session());

// Sign-in route
// Passport strategies are middlewares
app.post('/login', passport.authenticate('localSignin', {
    successRedirect: '/me',
    failureRedirect: '/login'
});

// Sign-up route
app.post('/register', passport.authenticate('localSignup', {
    successRedirect: '/',
    failureRedirect: '/signup'
});

// Call req.logout() to log out
app.get('/logout', function(req, res) {
    req.logout();
    res.redirect('/');
});

app.listen(3000);
```

Autenticazione Facebook

Il modulo **passport-facebook** viene utilizzato per implementare un'autenticazione di **Facebook** . In questo esempio, se l'utente non esiste al momento dell'accesso, viene creato.

Strategia di attuazione:

```
const passport = require('passport');
const FacebookStrategy = require('passport-facebook').Strategy;

// Strategy is named 'facebook' by default
passport.use({
    clientID: 'yourclientid',
    clientSecret: 'yourclientsecret',
    callbackURL: '/auth/facebook/callback'
},
// Facebook will send a token and user's profile
function(token, refreshToken, profile, next) {
    // Check in database if user is already registered
    findUserByFacebookId(profile.id, function(user) {
        // If user exists, returns his data to callback
        if (user) return next(null, user);
```

```
        // Else, we create the user
        else {
            let newUser = createUserFromFacebook(profile, token);

            newUser.save(function() {
                // Pass the user to the callback
                return next(null, newUser);
            });
        }
    });
});
```

Creare percorsi:

```
// ...
app.use(passport.initialize());
app.use(passport.session());

// Authentication route
app.get('/auth/facebook', passport.authenticate('facebook', {
    // Ask Facebook for more permissions
    scope : 'email'
}));

// Called after Facebook has authenticated the user
app.get('/auth/facebook/callback',
    passport.authenticate('facebook', {
        successRedirect : '/me',
        failureRedirect : '/'
}));


//...

app.listen(3000);
```

Semplice nome utente-password di autenticazione

Nei tuoi percorsi / index.js

Qui l' `user` è il modello per userSchema

```
router.post('/login', function(req, res, next) {
    if (!req.body.username || !req.body.password) {
        return res.status(400).json({
            message: 'Please fill out all fields'
        });
    }

    passport.authenticate('local', function(err, user, info) {
        if (err) {
            console.log("ERROR : " + err);
            return next(err);
        }

        if(user) (
```

```
            console.log("User Exists!")
            //All the data of the user can be accessed by user.x
            res.json({"success" : true});
            return;
        } else {
            res.json({"success" : false});
            console.log("Error" + errorResponse());
            return;
        }
    })(req, res, next);
});
```

Autenticazione Google Passport

Abbiamo modulo semplice disponibile in npm per goggle authetication name **passport-google-oauth20**

Considera l'esempio seguente In questo esempio abbiamo creato una cartella, ovvero config con i file passport.js e google.js nella directory root. Nella tua app.js include quanto segue

```
var express = require('express');
var session = require('express-session');
var passport = require('./config/passport'); // path where the passport file placed
var app = express();
passport(app);
```

// altro codice per inizializzare il server, handle di errore

Nel file passport.js nella cartella config è incluso il seguente codice

```
var passport = require  ('passport'),
google = require('./google'),
User = require('./../model/user'); // User is the  mongoose model

module.exports = function(app){
  app.use(passport.initialize());
  app.use(passport.session());
  passport.serializeUser(function(user, done){
      done(null, user);
  });
  passport.deserializeUser(function (user, done) {
     done(null, user);
  });
  google();
};
```

Di seguito il file google.js nella stessa cartella di configurazione

```
var passport = require('passport'),
GoogleStrategy = require('passport-google-oauth20').Strategy,
User = require('./../model/user');
module.exports = function () {
passport.use(new GoogleStrategy({
        clientID: 'CLIENT ID',
        clientSecret: 'CLIENT SECRET',
```

```
        callbackURL: "http://localhost:3000/auth/google/callback"
    },
    function(accessToken, refreshToken, profile, cb) {
        User.findOne({ googleId : profile.id }, function (err, user) {
            if(err){
                return cb(err, false, {message : err});
            }else {
                if (user != '' && user != null) {
                    return cb(null, user, {message : "User "});
                } else {
                    var username  = profile.displayName.split(' ');
                    var userData = new User({
                        name : profile.displayName,
                        username : username[0],
                        password : username[0],
                        facebookId : '',
                        googleId : profile.id,
                    });
                    // send email to user just in case required to send the newly created
                    // credentails to user for future login without using google login
                    userData.save(function (err, newuser) {
                        if (err) {
                            return cb(null, false, {message : err + " !!! Please try again"});
                        }else{
                            return cb(null, newuser);
                        }
                    });
                }
            }
        });
      }
  ));
};
```

Qui in questo esempio, se l'utente non è in DB, crea un nuovo utente in DB per riferimento locale utilizzando il nome campo googleId nel modello utente.

Leggi Integrazione del passaporto online: https://riptutorial.com/it/node-js/topic/7666/integrazione-del-passaporto

Capitolo 53: Integrazione di Cassandra

Examples

Ciao mondo

Per l'accesso `cassandra-driver` modulo Cassandra `cassandra-driver` Cassandra da DataStax può essere utilizzato. Supporta tutte le funzionalità e può essere facilmente configurato.

```
const cassandra = require("cassandra-driver");
const clientOptions = {
    contactPoints: ["host1", "host2"],
    keyspace: "test"
};

const client = new cassandra.Client(clientOptions);

const query = "SELECT hello FROM world WHERE name = ?";
client.execute(query, ["John"], (err, results) => {
    if (err) {
      return console.error(err);
    }

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

Leggi Integrazione di Cassandra online: https://riptutorial.com/it/node-js/topic/5949/integrazione-di-cassandra

Capitolo 54: Integrazione di PostgreSQL

Examples

Connetti a PostgreSQL

Utilizzo del modulo npm `PostgreSQL` .

installa la dipendenza da npm

```
npm  install pg --save
```

Ora devi creare una connessione PostgreSQL, che puoi in seguito interrogare.

Supponi tu Database_Name = studenti, Host = localhost e DB_User = postgres

```
var pg = require("pg")
var connectionString = "pg://postgres:postgres@localhost:5432/students";
var client = new pg.Client(connectionString);
client.connect();
```

Query con oggetto Connection

Se si desidera utilizzare l'oggetto di connessione per il database di query, è possibile utilizzare questo codice di esempio.

```
var queryString = "SELECT name, age FROM students " ;
var query = client.query(queryString);

query.on("row", (row, result)=> {
result.addRow(row);
});

query.on("end", function (result) {
//LOGIC
});
```

Leggi Integrazione di PostgreSQL online: https://riptutorial.com/it/node-js/topic/7706/integrazione-di-postgresql

Capitolo 55: Integrazione MongoDB per Node.js / Express.js

introduzione

MongoDB è uno dei più popolari database NoSQL, grazie all'aiuto dello stack MEAN. Interfacciare con un database Mongo da un'app Express è semplice e veloce, una volta compresa la sintassi della query. Useremo Mongoose per aiutarci.

Osservazioni

Ulteriori informazioni possono essere trovate qui: http://mongoosejs.com/docs/guide.html

Examples

Installare MongoDB

```
npm install --save mongodb
npm install --save mongoose //A simple wrapper for ease of development
```

Nel file del tuo server (normalmente denominato index.js o server.js)

```
const express = require('express');
const mongodb = require('mongodb');
const mongoose = require('mongoose');
const mongoConnectString = 'http://localhost/database name';

mongoose.connect(mongoConnectString, (err) => {
  if (err) {
    console.log('Could not connect to the database');
  }
});
```

Creazione di un modello Mongoose

```
const Schema = mongoose.Schema;
const ObjectId = Schema.Types.ObjectId;

const Article = new Schema({
  title: {
    type: String,
    unique: true,
    required: [true, 'Article must have title']
  },
  author: {
    type: ObjectId,
    ref: 'User'
```

```
  }
});

module.exports = mongoose.model('Article, Article);
```

Analizziamo questo. MongoDB e Mongoose usano JSON (in realtà BSON, ma qui è irrilevante) come formato dei dati. Nella parte superiore, ho impostato alcune variabili per ridurre la digitazione.

Creo un `new Schema` e lo assegno a una costante. È semplice JSON e ogni attributo è un altro oggetto con proprietà che consente di applicare uno schema più coerente. Forze univoche che le nuove istanze vengono inserite nel database, ovviamente, sono uniche. Questo è ottimo per evitare che un utente possa creare più account su un servizio.

Required è un altro, dichiarato come array. Il primo elemento è il valore booleano e il secondo il messaggio di errore se il valore inserito o aggiornato non esiste.

I ObjectID sono usati per le relazioni tra i Modelli. Gli esempi potrebbero essere "Gli utenti hanno molti commenti". Altri attributi possono essere usati al posto di ObjectId. Le stringhe come un nome utente sono un esempio.

Infine, l'esportazione del modello da utilizzare con le rotte API consente l'accesso al tuo schema.

Interrogare il tuo database Mongo

Una semplice richiesta GET. Supponiamo che il Modello dell'esempio precedente sia nel file `./db/models/Article.js` .

```
const express = require('express');
const Articles = require('./db/models/Article');

module.exports = function (app) {
  const routes = express.Router();

  routes.get('/articles', (req, res) => {
    Articles.find().limit(5).lean().exec((err, doc) => {
      if (doc.length > 0) {
        res.send({ data: doc });
      } else {
        res.send({ success: false, message: 'No documents retrieved' });
      }
    });
  });

app.use('/api', routes);
};
```

Ora possiamo ottenere i dati dal nostro database inviando una richiesta HTTP a questo endpoint. Alcune cose chiave, però:

1. Limit fa esattamente quello che sembra. Ricevo solo 5 documenti.
2. Lean elimina alcune cose dal BSON grezzo, riducendo la complessità e il sovraccarico. Non

richiesto. Ma utile

3. Quando usi `find` invece di `findOne` , conferma che `doc.length` è maggiore di 0. Questo perché `find` restituisce sempre un array, quindi un array vuoto non gestirà il tuo errore a meno che non venga controllato per la lunghezza
4. Personalmente mi piace inviare il messaggio di errore in quel formato. Modificalo in base alle tue esigenze. Stessa cosa per il documento restituito.
5. Il codice in questo esempio è stato scritto supponendo di averlo inserito in un altro file e non direttamente sul server Express. Per chiamare questo nel server, includere queste righe nel codice del server:

```
const app = express();
require('./path/to/this/file')(app) //
```

Leggi Integrazione MongoDB per Node.js / Express.js online: https://riptutorial.com/it/node-js/topic/9020/integrazione-mongodb-per-node-js---express-js

Capitolo 56: Interagire con la console

Sintassi

- console.log ([data] [, ...])
- console.error ([data] [, ...])
- console.time (etichetta)
- console.timeEnd (etichetta)

Examples

Registrazione

Modulo console

Simile all'ambiente browser di JavaScript node.js fornisce un modulo **console** che offre semplici possibilità di registrazione e debug.

I metodi più importanti forniti dal modulo della console sono `console.log` , `console.error` e `console.time` . Ma ce ne sono molti altri come `console.info` .

console.log

I parametri verranno stampati sullo standard output (`stdout`) con una nuova riga.

```
console.log('Hello World');
```

```
> console.log('Hello World')
Hello World
```

Console.Error

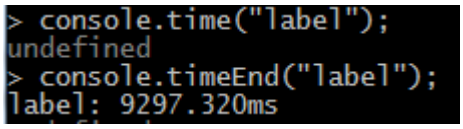
I parametri verranno stampati con l'errore standard (`stderr`) con una nuova riga.

```
console.error('Oh, sorry, there is an error.');
```

```
> console.error("Oh, sorry, error");
Oh, sorry, error
```

console.time, console.timeEnd

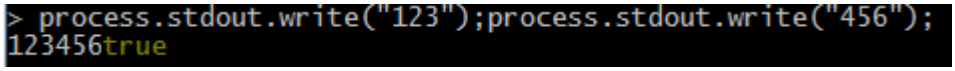
`console.time` avvia un timer con un'etichetta univoca che può essere utilizzata per calcolare la durata di un'operazione. Quando si chiama `console.timeEnd` con la stessa etichetta, il timer si arresta e stampa il tempo trascorso in millisecondi sullo `stdout` .



Modulo di processo

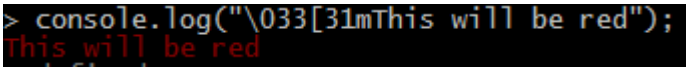
È possibile utilizzare il modulo di **processo** per scrivere **direttamente** nell'output standard della console. Pertanto esiste il metodo `process.stdout.write` . A differenza di `console.log` questo metodo non aggiunge una nuova riga prima dell'output.

Quindi nel seguente esempio il metodo viene chiamato due volte, ma nessuna nuova riga viene aggiunta tra i loro output.



formattazione

Si possono usare i **codici** di **terminale (controllo)** per emettere comandi specifici come cambiare i colori o posizionare il cursore.



Generale

Effetto	Codice
Reset	`\033[0m`
HiColor	`\033[1m`
Sottolineare	`\033[4m`
Inverso	`\033[7m`

Colori dei caratteri

Effetto	Codice
Nero	`\033[30m`
Rosso	`\033[31m`
verde	`\033[32m`

Effetto	Codice
Giallo	`\033[33m`
Blu	`\033[34m`
Magenta	`\033[35m`
Ciano	`\033[36m`
bianca	`\033[37m`

Colori di sfondo

Effetto	Codice
Nero	`\033[40m`
Rosso	`\033[41m`
verde	`\033[42m`
Giallo	`\033[43m`
Blu	`\033[44m`
Magenta	`\033[45m`
Ciano	`\033[46m`
bianca	`\033[47m`

Leggi Interagire con la console online: https://riptutorial.com/it/node-js/topic/5935/interagire-con-la-console

Capitolo 57: Invia notifica Web

Examples

Invia notifica Web utilizzando GCM (Google Cloud Messaging System)

Tale esempio sta conoscendo un'ampia diffusione tra **PWA** (Progressive Web Applications) e in questo esempio invieremo una semplice notifica di backend come usando **NodeJS** ed **ES6**

1. Installa il modulo Node-GCM: `npm install node-gcm`
2. Installa Socket.io: `npm install socket.io`
3. Crea un'applicazione GCM abilitata utilizzando Google Console.
4. Grabe il tuo ID applicazione GCM (ne avremo bisogno in seguito)
5. Inserisci il tuo codice segreto dell'applicazione GCM.
6. Apri il tuo editor di codice preferito e aggiungi il seguente codice:

```
'use strict';

const express = require('express');
const app = express();
const gcm = require('node-gcm');
app.io = require('socket.io')();

// [*] Configuring our GCM Channel.
const sender = new gcm.Sender('Project Secret');
const regTokens = [];
let message = new gcm.Message({
    data: {
        key1: 'msg1'
    }
});

// [*] Configuring our static files.
app.use(express.static('public/'));

// [*] Configuring Routes.
app.get('/', (req, res) => {
    res.sendFile(__dirname + '/public/index.html');
});

// [*] Configuring our Socket Connection.
app.io.on('connection', socket => {
    console.log('we have a new connection ...');
    socket.on('new_user', (reg_id) => {
        // [*] Adding our user notification registration token to our list typically
hided in a secret place.
        if (regTokens.indexOf(reg_id) === -1) {
            regTokens.push(reg_id);
```

```
            // [*] Sending our push messages
            sender.send(message, {
                registrationTokens: regTokens
            }, (err, response) => {
                if (err) console.error('err', err);
                else console.log(response);
            });
        }
    })
});

module.exports = app
```

PS: Sto usando un trucco speciale per far funzionare Socket.io con Express perché semplicemente non funziona fuori dagli schemi.

Ora creare un file **.json** e denominarlo: **Manifest.json** , aprirlo e **incollare** quanto segue:

```
{
    "name": "Application Name",
    "gcm_sender_id": "GCM Project ID"
}
```

Chiudilo e salva nella directory **ROOT** dell'applicazione.

PS: il file Manifest.json deve essere nella directory principale o non funzionerà.

Nel codice sopra sto facendo quanto segue:

1. Ho impostato e inviato una normale pagina index.html che utilizzerà anche socket.io.
2. Sto ascoltando un evento di **connessione** sparato dalla pagina **front-end** aka my **index.html** (verrà attivato una volta che un nuovo client si è connesso correttamente al nostro link predefinito)
3. Sto inviando un token speciale come il **token di registrazione** dal mio index.html tramite l'evento socket.io **new_user** , tale token sarà il nostro codice di accesso univoco dell'utente e ogni codice viene generato solitamente da un browser di supporto per l' **API di notifica Web** (leggi di più Qui.
4. Sto semplicemente usando il modulo **node-gcm** per inviare la mia notifica che verrà gestita e mostrata in seguito utilizzando i **Service Workers** `.

Questo è dal punto di vista di **NodeJS** . in altri esempi mostrerò come possiamo inviare dati personalizzati, icone .. etc nel nostro messaggio push.

PS: puoi trovare la demo completa di lavoro qui.

Leggi Invia notifica Web online: https://riptutorial.com/it/node-js/topic/6333/invia-notifica-web

Capitolo 58: Invio di un flusso di file al client

Examples

Utilizzo di fs And pipe per il flusso di file statici dal server

Un buon servizio VOD (Video On Demand) dovrebbe iniziare con le basi. Diciamo che hai una directory sul tuo server che non è pubblicamente accessibile, ma attraverso una sorta di portale o paywall vuoi consentire agli utenti di accedere ai tuoi file multimediali.

```
var movie = path.resolve('./public/' + req.params.filename);

        fs.stat(movie, function (err, stats) {

            var range = req.headers.range;

            if (!range) {

                return res.sendStatus(416);

            }

            //Chunk logic here
            var positions = range.replace(/bytes=/, "").split("-");
            var start = parseInt(positions[0], 10);
            var total = stats.size;
            var end = positions[1] ? parseInt(positions[1], 10) : total - 1;
            var chunksize = (end - start) + 1;

            res.writeHead(206, {

                'Transfer-Encoding': 'chunked',

                "Content-Range": "bytes " + start + "-" + end + "/" + total,

                "Accept-Ranges": "bytes",

                "Content-Length": chunksize,

                "Content-Type": mime.lookup(req.params.filename)

            });

            var stream = fs.createReadStream(movie, { start: start, end: end, autoClose: true
})

                .on('end', function () {

                    console.log('Stream Done');

                })

                .on("error", function (err) {

                    res.end(err);
```

```
            })
            .pipe(res, { end: true });
    });
```

Il frammento di cui sopra è uno schema di base per il modo in cui desideri trasmettere il tuo video in streaming a un cliente. La logica del blocco dipende da una varietà di fattori, tra cui il traffico di rete e la latenza. È importante bilanciare le dimensioni del mandrino rispetto alla quantità.

Infine, la chiamata .pipe consente a node.js di mantenere aperta una connessione con il server e di inviare chunk aggiuntivi, se necessario.

Streaming Utilizzando fluent-ffmpeg

Puoi anche usare flent-ffmpeg per convertire i file .mp4 in file .flv o altri tipi:

res.contentType ('flv');

```
    var pathToMovie = './public/' + req.params.filename;
    var proc = ffmpeg(pathToMovie)
        .preset('flashvideo')
        .on('end', function () {
            console.log('Stream Done');
        })
        .on('error', function (err) {
            console.log('an error happened: ' + err.message);
            res.send(err.message);
        })
        .pipe(res, { end: true });
```

Leggi Invio di un flusso di file al client online: https://riptutorial.com/it/node-js/topic/6994/invio-di-un-flusso-di-file-al-client

Capitolo 59: Koa Framework v2

Examples

Ciao esempio del mondo

```
const Koa = require('koa')

const app = new Koa()

app.use(async ctx => {
   ctx.body = 'Hello World'
})

app.listen(8080)
```

Gestione degli errori utilizzando il middleware

```
app.use(async (ctx, next) => {
  try {
    await next() // attempt to invoke the next middleware downstream
  } catch (err) {
    handleError(err, ctx) // define your own error handling function
  }
})
```

Leggi Koa Framework v2 online: https://riptutorial.com/it/node-js/topic/6730/koa-framework-v2

Capitolo 60: La gestione delle eccezioni

Examples

Gestione dell'eccezione in Node.Js

Node.js ha 3 modi di base per gestire eccezioni / errori:

1. **prova** - **cattura** blocco
2. **errore** come primo argomento di un `callback`
3. `emit` un evento di **errore** utilizzando eventEmitter

try-catch è usato per catturare le eccezioni generate dall'esecuzione del codice sincrono. Se il chiamante (o il chiamante del chiamante, ...) utilizza try / catch, allora possono rilevare l'errore. Se nessuno dei chiamanti ha tentato di intercettarlo, il programma si blocca.
Se si utilizza try-catch su un'operazione asincrona, è stata generata un'eccezione dal richiamo del metodo asincrono che non verrà catturata da try-catch. Per catturare un'eccezione dalla richiamata dell'operazione asincrona, è preferibile utilizzare le *promesse* .
Esempio per capirlo meglio

```
// ** Example - 1  **
function doSomeSynchronousOperation(req, res) {
    if(req.body.username === ''){
        throw new Error('User Name cannot be empty');
    }
    return true;
}

// calling the method above
try {
    // synchronous code
    doSomeSynchronousOperation(req, res)
catch(e) {
    //exception handled here
    console.log(e.message);
}

// ** Example - 2 **
function doSomeAsynchronousOperation(req, res, cb) {
    // imitating async operation
    return setTimeout(function(){
        cb(null, []);
    },1000);
}

try {
    // asynchronous code
    doSomeAsynchronousOperation(req, res, function(err, rs){
        throw new Error("async operation exception");
    })
} catch(e) {
     // Exception will not get handled here
     console.log(e.message);
```

```
}
// The exception is unhandled and hence will cause application to break
```

le callback sono utilizzate principalmente in Node.js in quanto il callback consegna un evento in modo asincrono. L'utente ti passa una funzione (il callback) e la invochi più tardi quando l'operazione asincrona viene completata.
Lo schema normale è che la richiamata è invocata come *callback (err, risultato)* , dove solo uno di err e result è non nullo, a seconda che l'operazione abbia avuto esito positivo o negativo.

```
function doSomeAsynchronousOperation(req, res, callback) {
   setTimeout(function(){
        return callback(new Error('User Name cannot be empty'));
   }, 1000);
   return true;
}

doSomeAsynchronousOperation(req, res, function(err, result) {
   if (err) {
       //exception handled here
       console.log(err.message);
   }

   //do some stuff with valid data
});
```

emetta Per i casi più complicati, invece di utilizzare una richiamata, la funzione stessa può restituire un oggetto EventEmitter e il chiamante dovrebbe ascoltare gli eventi di errore sull'emettitore.

```
const EventEmitter = require('events');

function doSomeAsynchronousOperation(req, res) {
    let myEvent = new EventEmitter();

    // runs asynchronously
    setTimeout(function(){
        myEvent.emit('error', new Error('User Name cannot be empty'));
    }, 1000);

    return myEvent;
}

// Invoke the function
let event = doSomeAsynchronousOperation(req, res);

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

event.on('done', function(result) {
    console.log(result); // true
});
```

Gestione delle eccezioni non gestita

Poiché Node.js viene eseguito su un singolo processo, le eccezioni non rilevate sono un problema da tenere presente quando si sviluppano applicazioni.

Eccezioni di gestione silenziosa

La maggior parte delle persone lascia che i server node.js soppongano silenziosamente gli errori.

- Gestendo in silenzio l'eccezione

```
process.on('uncaughtException', function (err) {
  console.log(err);
});
```

Questo è male , funzionerà ma:

- La causa principale rimarrà sconosciuta, in quanto tale non contribuirà alla risoluzione di ciò che ha causato l'eccezione (errore).
- Nel caso in cui la connessione al database (pool) venga chiusa per qualche motivo, ciò comporterà una costante propagazione degli errori, il che significa che il server sarà in esecuzione ma non si riconnetterà a db.

Ritorno allo stato iniziale

In caso di "uncaughtException" è bene riavviare il server e riportarlo allo **stato iniziale** , dove sappiamo che funzionerà. Viene registrata un'eccezione, l'applicazione viene chiusa ma, poiché verrà eseguita in un contenitore che si accerterà che il server sia in esecuzione, otterremo il riavvio del server (ritorno allo stato operativo iniziale).

- Installazione per sempre (o altro strumento CLI per assicurarsi che quel server nodo funzioni continuamente)

```
npm install forever -g
```

- Avvio del server per sempre

```
forever start app.js
```

  Il motivo per cui è stato avviato e il motivo per cui utilizziamo per sempre è dopo che il server è stato **interrotto** per sempre. Il processo riavvierà il server.

- Riavvio del server

```
process.on('uncaughtException', function (err) {
    console.log(err);

    // some logging mechanisam
```

```
    // ....

    process.exit(1); // terminates process
});
```

In una nota a margine c'era anche un modo per gestire le eccezioni con **Cluster e Domini** .

I domini sono deprecati qui più informazioni.

Errori e promesse

Le promesse gestiscono gli errori in modo diverso rispetto al codice sincrono o callback.

```
const p = new Promise(function (resolve, reject) {
    reject(new Error('Oops'));
});

// anything that is `reject`ed inside a promise will be available through catch
// while a promise is rejected, `.then` will not be called
p
    .then(() => {
        console.log("won't be called");
    })
    .catch(e => {
        console.log(e.message); // output: Oops
    })
    // once the error is caught, execution flow resumes
    .then(() => {
        console.log('hello!'); // output: hello!
    });
```

attualmente, gli errori generati da una promessa che non viene rilevata provocano l'errore di essere ingeriti, il che può rendere difficile rintracciare l'errore. Questo può essere risolto utilizzando strumenti di linting come eslint o assicurando di avere sempre una clausola di `catch` .

Questo comportamento è deprecato nel nodo 8 a favore del termine del processo del nodo.

Leggi La gestione delle eccezioni online: https://riptutorial.com/it/node-js/topic/2819/la-gestione-delle-eccezioni

Capitolo 61: Le notifiche push

introduzione

Quindi, se vuoi fare una notifica per le app Web, ti suggerisco di utilizzare il framework Push.js o SoneSignal per l'app Web / mobile.

Push è il modo più veloce per essere subito operativo con le notifiche Javascript. Una nuova aggiunta alle specifiche ufficiali, l'API di notifica consente ai browser moderni come Chrome, Safari, Firefox e IE 9+ di inviare notifiche al desktop di un utente.

Dovrai usare Socket.io e qualche framework di backend, userò Express per questo esempio.

Parametri

Modulo / quadro	descrizione
node.js / espresso	Semplice framework back-end per l'applicazione Node.js, molto facile da usare ed estremamente potente
Socket.io	Socket.IO consente comunicazioni bidirezionali basate su eventi in tempo reale. Funziona su ogni piattaforma, browser o dispositivo, concentrandosi in egual misura su affidabilità e velocità.
Push.js	Il framework di notifiche desktop più versatile al mondo
OneSignal	Solo un altro modulo di notifiche push per dispositivi Apple
Firebase	Firebase è la piattaforma mobile di Google che ti aiuta a sviluppare rapidamente app di alta qualità e a far crescere la tua attività.

Examples

Notifica Web

Innanzitutto, devi installare il modulo Push.js.

```
$ npm install push.js --save
```

Oppure importalo nella tua app front-end tramite CDN

```
<script src="./push.min.js"></script> <!-- CDN link -->
```

Dopo aver finito con quello, dovresti essere bravo ad andare. Ecco come dovrebbe apparire se vuoi fare una semplice notifica:

```
Push.create('Hello World!')
```

Presumo che tu sappia come configurare Socket.io con la tua app. Ecco alcuni esempi di codice della mia app di backend con express:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);

server.listen(80);

app.get('/', function (req, res) {
  res.sendfile(__dirname + '/index.html');
});

io.on('connection', function (socket) {

  socket.emit('pushNotification', { success: true, msg: 'hello' });

});
```

Dopo aver configurato il tuo server, dovresti essere in grado di passare a cose di front-end. Ora tutto ciò che dobbiamo fare è importare Socket.io CDN e aggiungere questo codice al mio file *index.html* :

```
<script src="../socket.io.js"></script> <!-- CDN link -->
<script>
  var socket = io.connect('http://localhost');
  socket.on('pushNotification', function (data) {
    console.log(data);
    Push.create("Hello world!", {
        body: data.msg, //this should print "hello"
        icon: '/icon.png',
        timeout: 4000,
        onClick: function () {
            window.focus();
            this.close();
        }
    });
  });
</script>
```

Ecco fatto, ora dovresti essere in grado di visualizzare la tua notifica, questo funziona anche su qualsiasi dispositivo Android, e se vuoi usare la messaggistica cloud Firebase , puoi usarlo con questo modulo, Ecco il link per quell'esempio scritto da Nick (creatore di Push.js)

Mela

Tieni presente che ciò non funzionerà sui dispositivi Apple (non li ho testati tutti), ma se vuoi fare notifiche push controlla il plug-in OneSignal .

Leggi Le notifiche push online: https://riptutorial.com/it/node-js/topic/10892/le-notifiche-push

Capitolo 62: Linea di lettura

Sintassi

- const readline = require ('readline')
- readline.close ()
- readline.pause ()
- readline.prompt ([preserveCursor])
- readline.question (query, callback)
- readline.resume ()
- readline.setPrompt (prompt)
- readline.write (data [, chiave])
- readline.clearLine (stream, dir)
- readline.clearScreenDown (stream)
- readline.createInterface (opzioni)
- readline.cursorTo (stream, x, y)
- readline.emitKeypressEvents (stream [, interfaccia])
- readline.moveCursor (stream, dx, dy)

Examples

Lettura file riga per riga

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

const rl = readline.createInterface({
    input: fs.createReadStream('text.txt')
});

// Each new line emits an event - every time the stream receives \r, \n, or \r\n
rl.on('line', (line) => {
    console.log(line);
});

rl.on('close', () => {
    console.log('Done reading file');
});
```

Richiesta di input da parte dell'utente tramite CLI

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

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question('What is your name?', (name) => {
```

```
    console.log(`Hello ${name}!`);

    rl.close();
});
```

Leggi Linea di lettura online: https://riptutorial.com/it/node-js/topic/1431/linea-di-lettura

Capitolo 63: Localizzazione del nodo JS

introduzione

È molto semplice da gestire nodejs di localizzazione express

Examples

utilizzando il modulo i18n per mantenere la localizzazione nell'app nodo js

Modulo di traduzione semplice e leggero con memoria json dinamica. Supporta semplici applicazioni node.js di vanilla e dovrebbe funzionare con qualsiasi framework (come express, restify e probabilmente più) che espone un metodo app.use () che passa in oggetti res e req. Utilizza la sintassi comune __ ('...') in app e modelli. Memorizza i file di lingua in file json compatibili con il formato webtranslateit json. Aggiunge nuove stringhe al volo quando vengono utilizzate per la prima volta nella tua app. Non è necessaria alcuna analisi extra.

express + i18n-node + cookieParser ed evita problemi di concorrenza

```
// usual requirements
var express = require('express'),
    i18n = require('i18n'),
    app = module.exports = express();

i18n.configure({
  // setup some locales - other locales default to en silently
  locales: ['en', 'ru', 'de'],

  // sets a custom cookie name to parse locale settings from
  cookie: 'yourcookiename',

  // where to store json files - defaults to './locales'
  directory: __dirname + '/locales'
});

app.configure(function () {
  // you will need to use cookieParser to expose cookies to req.cookies
  app.use(express.cookieParser());

  // i18n init parses req for language headers, cookies, etc.
  app.use(i18n.init);

});

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

// starting server
if (!module.parent) {
  app.listen(3000);
```

```
}
```

Leggi Localizzazione del nodo JS online: https://riptutorial.com/it/node-js/topic/9594/localizzazione-del-nodo-js

Capitolo 64: Lodash

introduzione

Lodash è una comoda libreria di utilità JavaScript.

Examples

Filtra una raccolta

Il frammento di codice qui sotto mostra i vari modi in cui puoi filtrare su una serie di oggetti usando lodash.

```
let lodash = require('lodash');

var countries = [
    {"key": "DE", "name": "Deutschland", "active": false},
    {"key": "ZA", "name": "South Africa", "active": true}
];

var filteredByFunction = lodash.filter(countries, function (country) {
    return country.key === "DE";
});
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByObjectProperties = lodash.filter(countries, { "key": "DE" });
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByProperties = lodash.filter(countries, ["key", "ZA"]);
// => [{"key": "ZA", "name": "South Africa"}];

var filteredByProperty = lodash.filter(countries, "active");
// => [{"key": "ZA", "name": "South Africa"}];
```

Leggi Lodash online: https://riptutorial.com/it/node-js/topic/9161/lodash

Capitolo 65: Loopback - Connettore basato REST

introduzione

Connettori basati sul riposo e come gestirli. Sappiamo tutti che Loopback non fornisce eleganza alle connessioni basate su REST

Examples

Aggiungere un connettore basato sul web

```
// Questo esempio ottiene la risposta da iTunes
{
  "riposo": {
    "nome": "resto",
    "connettore": "resto",
    "debug": vero,
    "opzioni": {
      "useQuerystring": true,
      "timeout": 10000,
      "intestazioni": {
        "accetta": "application / json",
        "content-type": "application / json"
      }
    },
    "operazioni": [
      {
        "modello": {
          "metodo": "OTTIENI",
          "url": "https://itunes.apple.com/search",
          "query": {
            "termine": "{parola chiave}",
            "Paese": "{country = IN}",
            "media": "{itemType = music}",
            "limit": "{limit = 10}",
            "esplicito": "falso"
          }
        },
        "funzioni": {
          "ricerca": [
            "parola chiave",
            "nazione",
            "tipo di elemento",
            "limite"
          ]
        }
      },
      {
        "modello": {
          "metodo": "OTTIENI",
          "url": "https://itunes.apple.com/lookup",
          "query": {
```

```
            "L'ho fatto}"
          }
        },
        "funzioni": {
          "findById": [
            "Id"
          ]
        }
      }
    ]
  }
}
```

Leggi Loopback - Connettore basato REST online: https://riptutorial.com/it/node-js/topic/9234/loopback---connettore-basato-rest

Capitolo 66: Mantenere costantemente attiva un'applicazione di nodo

Examples

Utilizzare PM2 come gestore processi

PM2 ti consente di eseguire gli script nodejs per sempre. Nel caso in cui l'applicazione si arresti in modo anomalo, anche PM2 lo riavvierà automaticamente.

Installa PM2 a livello globale per gestire le istanze nodejs

```
npm install pm2 -g
```

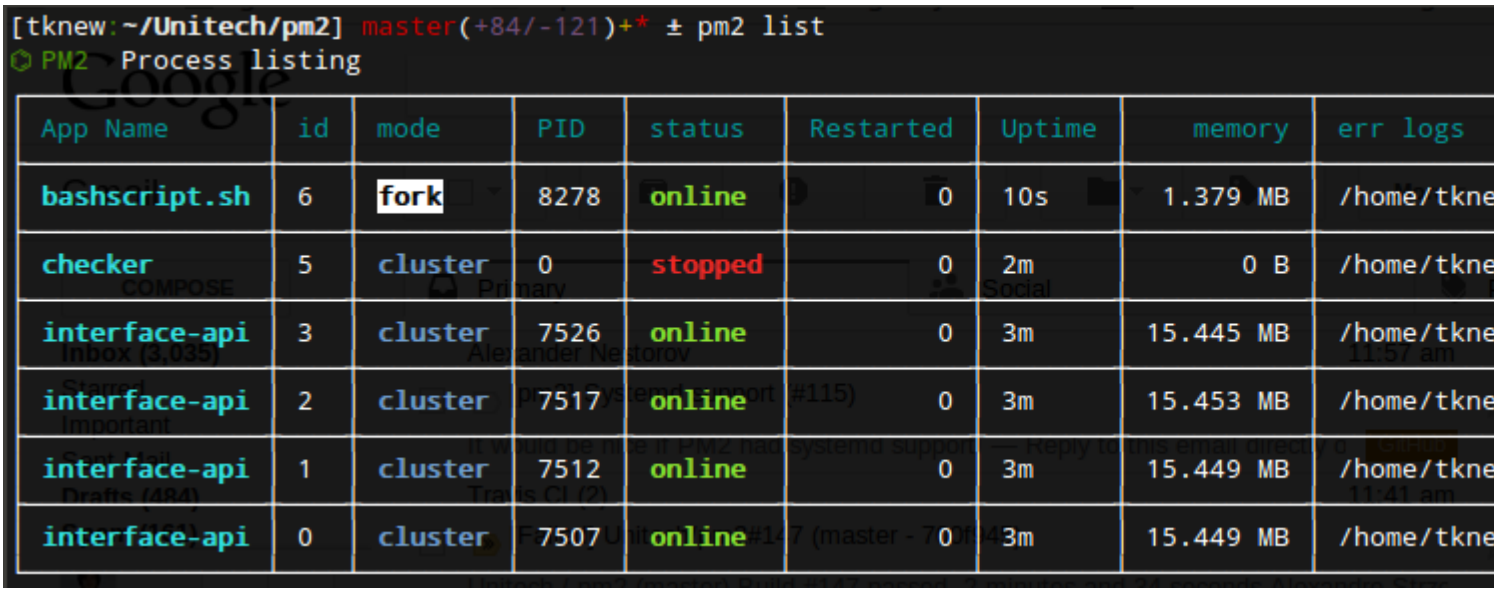
Passare alla directory in cui risiede lo script nodejs ed eseguire il seguente comando ogni volta che si desidera avviare un'istanza nodejs da monitorare tramite pm2:

```
pm2 start server.js --name "app1"
```

Comandi utili per il monitoraggio del processo

1. Elenca tutte le istanze nodejs gestite da pm2

   ```
   pm2 list
   ```



2. Arresta una particolare istanza nodejs

   ```
   pm2 stop <instance named>
   ```

3. Elimina una particolare istanza nodejs

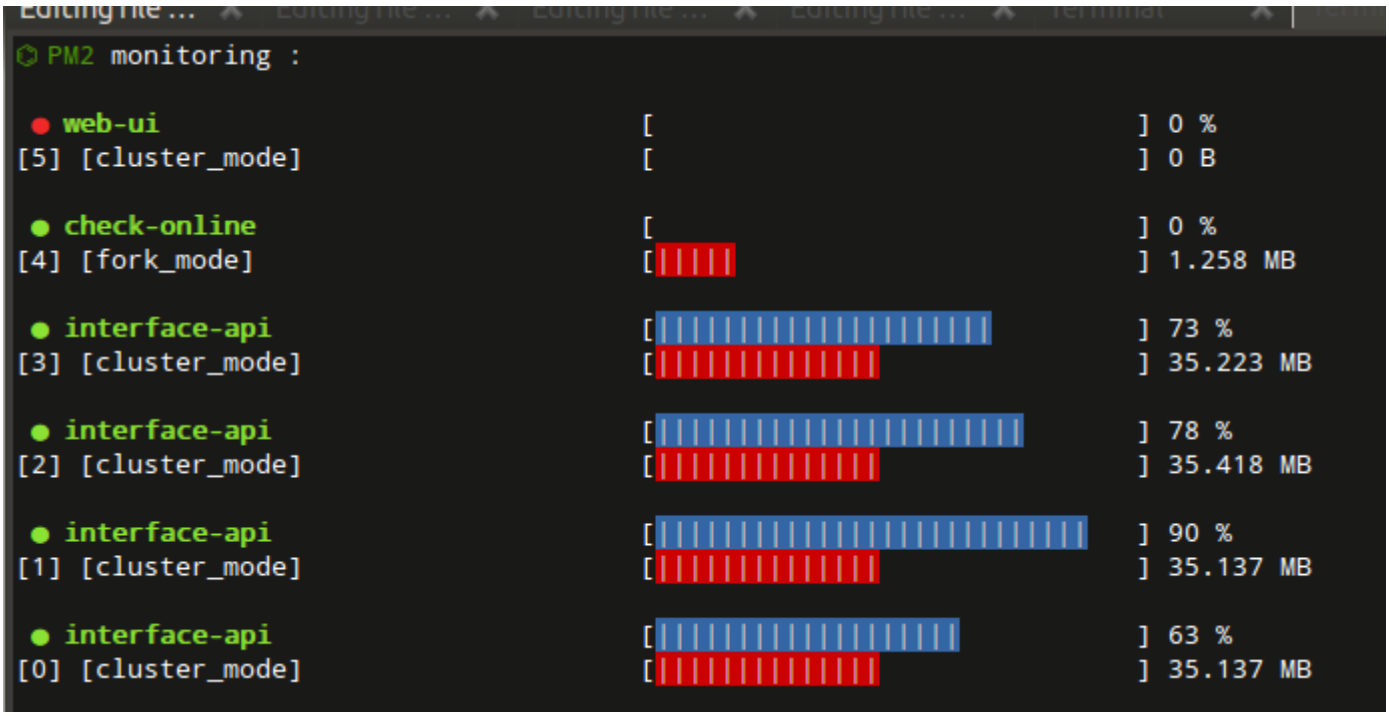
```
pm2 delete <instance name>
```

4. Riavvia una particolare istanza nodejs

```
pm2 restart <instance name>
```

5. Monitoraggio di tutte le istanze nodejs

```
pm2 monit
```



6. Stop pm2

```
pm2 kill
```

7. A differenza del riavvio, che uccide e riavvia il processo, ricarica un tempo di ricarica di 0 secondi

```
pm2 reload <instance name>
```

8. Visualizza i log

```
pm2 logs <instance_name>
```

Esecuzione e arresto di un daemon Forever

Per iniziare il processo:

```
$ forever start index.js
warn:    --minUptime not set. Defaulting to: 1000ms
warn:    --spinSleepTime not set. Your script will exit if it does not stay up for at least
1000ms
info:    Forever processing file: index.js
```

Elenco in esecuzione per sempre istanze:

```
$ forever list
info:    Forever processes running

|data: | index | uid | command          | script      |forever pid|id    | logfile
|uptime        |
|------|-------|-----|------------------|-------------|-----------|-----|---------------------
---|--------------|
|data: | [0]   |f4Kt |/usr/bin/nodejs   | src/index.js|2131       |
2146|/root/.forever/f4Kt.log | 0:0:0:11.485 |
```

Fermare il primo processo:

```
$ forever stop 0

$ forever stop 2146

$ forever stop --uid f4Kt

$ forever stop --pidFile 2131
```

Funzionamento continuo con nohup

Un'alternativa a per sempre su Linux è nohup.

Per avviare un'istanza nohup

1. cd nella posizione di `app.js` o cartella `www`
2. eseguire `nohup nodejs app.js &`

Per uccidere il processo

1. lanciare `ps -ef|grep nodejs`
2. `kill -9 <the process number>`

Process Mangement with Forever

Installazione

```
npm install forever -g
cd /node/project/directory
```

usi

```
forever start app.js
```

Leggi Mantenere costantemente attiva un'applicazione di nodo online: https://riptutorial.com/it/node-js/topic/2820/mantenere-costantemente-attiva-un-applicazione-di-nodo

Capitolo 67: metalsmith

Examples

Costruisci un semplice blog

Supponendo di avere node e npm installati e disponibili, creare una cartella di progetto con un `package.json` . Installa le dipendenze necessarie:

```
npm install --save-dev metalsmith metalsmith-in-place handlebars
```

Creare un file chiamato `build.js` nella `build.js` principale della cartella del progetto, contenente quanto segue:

```
var metalsmith = require('metalsmith');
var handlebars = require('handlebars');
var inPlace = require('metalsmith-in-place');

Metalsmith(__dirname)
  .use(inPlace('handlebars'))
  .build(function(err) {
    if (err) throw err;
    console.log('Build finished!');
  });
```

Crea una cartella chiamata `src` nella radice della cartella del tuo progetto. Crea `index.html` in `src` , contenente quanto segue:

```
---
title: My awesome blog
---
<h1>{{ title }}</h1>
```

L'esecuzione di `node build.js` ora creerà tutti i file in `src` . Dopo aver eseguito questo comando, avrai `index.html` nella tua cartella di compilazione, con i seguenti contenuti:

```
<h1>My awesome blog</h1>
```

Leggi metalsmith online: https://riptutorial.com/it/node-js/topic/6111/metalsmith

Capitolo 68: Modulo Cluster

Sintassi

- const cluster = require ("cluster")
- cluster.fork ()
- cluster.isMaster
- cluster.isWorker
- cluster.schedulingPolicy
- cluster.setupMaster (impostazioni)
- cluster.settings
- cluster.worker // in worker
- cluster.workers // in master

Osservazioni

Si noti che `cluster.fork()` genera un processo figlio che inizia a eseguire lo script corrente dall'inizio, in contrasto con la chiamata di sistema `fork()` in *C* che clona il processo corrente e continua dall'istruzione dopo la chiamata di sistema in entrambi i genitori e processo figlio.

La documentazione di Node.js ha una guida più completa ai cluster qui

Examples

Ciao mondo

Questo è il tuo `cluster.js` :

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  require('./server.js')();
}
```

Questo è il tuo `server.js` principale:

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

function startServer() {
    const server = http.createServer((req, res) => {
      res.writeHead(200);
      res.end('Hello Http');
    });

    server.listen(3000);
}

if(!module.parent) {
    // Start server if file is run directly
    startServer();
} else {
    // Export server, if file is referenced via cluster
    module.exports = startServer;
}
```

In questo esempio, ospitiamo un server Web di base, tuttavia, giriamo i worker (processi secondari) utilizzando il modulo **cluster** incorporato. Il numero di processi forker dipende dal numero di core CPU disponibili. Ciò consente a un'applicazione Node.js di sfruttare le CPU multi-core, poiché una singola istanza di Node.js viene eseguita in un singolo thread. L'applicazione condividerà ora la porta 8000 su tutti i processi. I carichi verranno automaticamente distribuiti tra i lavoratori utilizzando il metodo Round-Robin per impostazione predefinita.

Esempio di cluster

Una singola istanza di `Node.js` viene eseguita in un singolo thread. Per sfruttare i sistemi multi-core, l'applicazione può essere avviata in un cluster di processi Node.js per gestire il carico.

Il modulo `cluster` consente di creare facilmente processi figlio che condividono tutte le porte del server.

L'esempio seguente crea il processo child worker nel processo principale che gestisce il carico su più core.

Esempio

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length; //number of CPUS

if (cluster.isMaster) {
  // Fork workers.
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();    //creating child process
  }

  //on exit of cluster
  cluster.on('exit', (worker, code, signal) => {
      if (signal) {
        console.log(`worker was killed by signal: ${signal}`);
      } else if (code !== 0) {
        console.log(`worker exited with error code: ${code}`);
```

```
      } else {
        console.log('worker success!');
      }
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(3000);
}
```

Leggi Modulo Cluster online: https://riptutorial.com/it/node-js/topic/2817/modulo-cluster

Capitolo 69: MSSQL Intergration

introduzione

Per integrare qualsiasi database con nodejs è necessario un pacchetto driver o si può chiamare un modulo npm che fornirà API di base per connettersi con il database ed eseguire interazioni. Lo stesso vale per il database mssql, qui integreremo mssql con nodejs ed eseguiremo alcune query di base su tabelle SQL.

Osservazioni

Abbiamo assunto che avremo un'istanza locale del server di database mssql in esecuzione sul computer locale. Puoi fare riferimento a questo documento per fare lo stesso.

Assicurati inoltre di aver aggiunto l'utente appropriato creato con i privilegi.

Examples

Connessione con SQL tramite. modulo mssql npm

Inizieremo con la creazione di una semplice applicazione di nodo con una struttura di base e quindi la connessione con il database del server SQL locale e l'esecuzione di alcune query su quel database.

Passaggio 1: creare una directory / cartella in base al nome del progetto che si intende creare. Inizializza un'applicazione di nodo utilizzando il comando *npm init* che creerà un package.json nella directory corrente.

```
mkdir mySqlApp
//folder created
cd mwSqlApp
//change to newly created directory
npm init
//answer all the question ..
npm install
//This will complete quickly since we have not added any packages to our app.
```

Passo 2: Ora creeremo un file App.js in questa directory e installeremo alcuni pacchetti che dovranno essere collegati a sql db.

```
sudo gedit App.js
//This will create App.js file , you can use your fav. text editor :)
npm install --save mssql
//This will install the mssql package to you app
```

Passo 3: Ora aggiungeremo alla nostra applicazione una variabile di configurazione di base che verrà utilizzata dal modulo mssql per stabilire una connessione.

```
console.log("Hello world, This is an app to connect to sql server.");
var config = {
        "user": "myusername", //default is sa
        "password": "yourStrong(!)Password",
        "server": "localhost", // for local machine
        "database": "staging", // name of database
        "options": {
            "encrypt": true
        }
      }

sql.connect(config, err => {
    if(err){
        throw err ;
    }
    console.log("Connection Successful !");

    new sql.Request().query('select 1 as number', (err, result) => {
        //handle err
        console.dir(result)
        // This example uses callbacks strategy for getting results.
    })

});

sql.on('error', err => {
    // ... error handler
    console.log("Sql database connection error " ,err);
})
```

Passaggio 4: questo è il passaggio più semplice, in cui avviamo l'applicazione e l'applicazione si connetterà al server sql e stamperà alcuni risultati semplici.

```
node App.js
// Output :
// Hello world, This is an app to connect to sql server.
// Connection Successful !
// 1
```

> Per utilizzare le promesse o asincrone per l'esecuzione della query, fare riferimento ai documenti ufficiali del pacchetto mssql:
>
> - promesse
> - Async / Await

Leggi MSSQL Intergration online: https://riptutorial.com/it/node-js/topic/9884/mssql-intergration

Capitolo 70: multithreading

introduzione

Node.js è stato progettato per essere thread singolo. Quindi, per tutti gli scopi pratici, le applicazioni che si avviano con Node verranno eseguite su un singolo thread.

Tuttavia, Node.js stesso esegue multi-thread. Le operazioni di I / O e simili verranno eseguite da un pool di thread. Inoltre, qualsiasi istanza di un'applicazione nodo viene eseguita su un thread diverso, pertanto per eseguire applicazioni multi-thread si avviano più istanze.

Osservazioni

Capire il Loop degli Eventi è importante per capire come e perchè usare più thread.

Examples

Grappolo

Il modulo `cluster` consente di avviare la stessa applicazione più volte.

Il clustering è auspicabile quando le diverse istanze hanno lo stesso flusso di esecuzione e non dipendono l'una dall'altra. In questo scenario, hai un master che può avviare fork e fork (o figli). I bambini lavorano indipendentemente e hanno il loro unico spazio di Ram ed Event Loop.

La configurazione dei cluster può essere utile per i siti Web / le API. Qualsiasi thread può servire qualsiasi cliente, in quanto non dipende da altri thread. Un database (come Redis) verrebbe utilizzato per condividere cookie, in quanto le **variabili non possono essere condivise!** tra i fili.

```
// runs in each instance
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;

console.log('I am always called');

if (cluster.isMaster) {
    // runs only once (within the master);
    console.log('I am the master, launching workers!');
    for(var i = 0; i < numCPUs; i++) cluster.fork();

} else {
    // runs in each fork
    console.log('I am a fork!');

    // here one could start, as an example, a web server

}

console.log('I am always called as well');
```

Processo figlio

I processi figlio sono la strada da seguire quando si desidera eseguire processi in modo indipendente con diverse inizializzazioni e preoccupazioni. Come i fork nei cluster, un `child_process` viene eseguito nel thread, ma a differenza dei fork, ha un modo di comunicare con i suoi genitori.

La comunicazione va in entrambe le direzioni, quindi genitore e figlio possono ascoltare i messaggi e inviare messaggi.

Parent (../parent.js)

```
var child_process = require('child_process');
console.log('[Parent]', 'initalize');

var child1 = child_process.fork(__dirname + '/child');
child1.on('message', function(msg) {
    console.log('[Parent]', 'Answer from child: ', msg);
});

// one can send as many messages as one want
child1.send('Hello'); // Hello to you too :)
child1.send('Hello'); // Hello to you too :)

// one can also have multiple children
var child2 = child_process.fork(__dirname + '/child');
```

Bambino (../child.js)

```
// here would one initialize this child
// this will be executed only once
console.log('[Child]', 'initalize');

// here one listens for new tasks from the parent
process.on('message', function(messageFromParent) {

    //do some intense work here
    console.log('[Child]', 'Child doing some intense work');

    if(messageFromParent == 'Hello') process.send('Hello to you too :)');
    else process.send('what?');

})
```

Accanto al messaggio si possono ascoltare molti eventi come "errore", "connesso" o "disconnessione".

L'avvio di un processo secondario comporta un certo costo associato. Si vorrebbe generare il maggior numero possibile di loro.

Leggi multithreading online: https://riptutorial.com/it/node-js/topic/10592/multithreading

Capitolo 71: N-API

introduzione

La N-API è un modo nuovo e migliore per creare un modulo nativo per NodeJS. N-API è nella fase iniziale in modo che possa avere documentazione incoerente.

Examples

Ciao a N-API

Questo modulo registra la funzione ciao sul modulo hello. ciao funzione stampa Hello world su console con `printf` e restituisce `1373` dalla funzione nativa nel chiamante javascript.

```
#include <node_api.h>
#include <stdio.h>


napi_value say_hello(napi_env env, napi_callback_info info)
{
    napi_value retval;

    printf("Hello world\n");

    napi_create_number(env, 1373, &retval);

    return retval;
}

void init(napi_env env, napi_value exports, napi_value module, void* priv)
{
    napi_status status;
    napi_property_descriptor desc = {
        /*
         * String describing the key for the property, encoded as UTF8.
         */
        .utf8name = "hello",
        /*
         * Set this to make the property descriptor object's value property
         * to be a JavaScript function represented by method.
         * If this is passed in, set value, getter and setter to NULL (since these members
won't be used).
         */
        .method = say_hello,
        /*
         * A function to call when a get access of the property is performed.
         * If this is passed in, set value and method to NULL (since these members won't be
used).
         * The given function is called implicitly by the runtime when the property is
accessed
         * from JavaScript code (or if a get on the property is performed using a N-API call).
         */
        .getter = NULL,
        /*
```

```
         * A function to call when a set access of the property is performed.
         * If this is passed in, set value and method to NULL (since these members won't be
used).
         * The given function is called implicitly by the runtime when the property is set
         * from JavaScript code (or if a set on the property is performed using a N-API call).
         */
        .setter = NULL,
        /*
         * The value that's retrieved by a get access of the property if the property is a
data property.
         * If this is passed in, set getter, setter, method and data to NULL (since these
members won't be used).
         */
        .value = NULL,
        /*
         * The attributes associated with the particular property. See
napi_property_attributes.
         */
        .attributes = napi_default,
        /*
         * The callback data passed into method, getter and setter if this function is
invoked.
         */
        .data = NULL
    };
    /*
     * This method allows the efficient definition of multiple properties on a given object.
     */
    status = napi_define_properties(env, exports, 1, &desc);

    if (status != napi_ok)
        return;
}


NAPI_MODULE(hello, init)
```

Leggi N-API online: https://riptutorial.com/it/node-js/topic/10539/n-api

Capitolo 72: Node server senza framework

Osservazioni

Sebbene Node abbia molti framework per aiutarti a far funzionare il tuo server, principalmente:

Express : la struttura più utilizzata

Totale : il framework ALL-IN-ONE UNITY, che ha tutto e non dipende da nessun altro framework o modulo.

Tuttavia, non esiste sempre una taglia adatta a tutti, quindi lo sviluppatore potrebbe aver bisogno di creare il proprio server, senza altre dipendenze.

Se l'app a cui si accede tramite un server esterno, CORS potrebbe essere un problema, un codice per evitare che fosse stato fornito.

Examples

Node server senza Framework

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

http.createServer(function (request, response) {
console.log('request ', request.url);

var filePath = '.' + request.url;
if (filePath == './')
    filePath = './index.html';

var extname = String(path.extname(filePath)).toLowerCase();
var contentType = 'text/html';
var mimeTypes = {
    '.html': 'text/html',
    '.js': 'text/javascript',
    '.css': 'text/css',
    '.json': 'application/json',
    '.png': 'image/png',
    '.jpg': 'image/jpg',
    '.gif': 'image/gif',
    '.wav': 'audio/wav',
    '.mp4': 'video/mp4',
    '.woff': 'application/font-woff',
    '.ttf': 'applilcation/font-ttf',
    '.eot': 'application/vnd.ms-fontobject',
    '.otf': 'application/font-otf',
    '.svg': 'application/image/svg+xml'
};

contentType = mimeTypes[extname] || 'application/octect-stream';
```

```
fs.readFile(filePath, function(error, content) {
    if (error) {
        if(error.code == 'ENOENT'){
            fs.readFile('./404.html', function(error, content) {
                response.writeHead(200, { 'Content-Type': contentType });
                response.end(content, 'utf-8');
            });
        }
        else {
            response.writeHead(500);
            response.end('Sorry, check with the site admin for error: '+error.code+' ..\n');
            response.end();
        }
    }
    else {
        response.writeHead(200, { 'Content-Type': contentType });
        response.end(content, 'utf-8');
    }
 });

}).listen(8125);
console.log('Server running at http://127.0.0.1:8125/');
```

Superare i problemi CORS

```
// Website you wish to allow to connect to
response.setHeader('Access-Control-Allow-Origin', '*');

// Request methods you wish to allow
response.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE');

// Request headers you wish to allow
response.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');

// Set to true if you need the website to include cookies in the requests sent
// to the API (e.g. in case you use sessions)
response.setHeader('Access-Control-Allow-Credentials', true);
```

Leggi Node server senza framework online: https://riptutorial.com/it/node-js/topic/5910/node-server-senza-framework

Capitolo 73: Node.js (express.js) con il codice di esempio angular.js

introduzione

Questo esempio mostra come creare un'applicazione Express di base e quindi servire AngularJS.

Examples

Creare il nostro progetto.

Siamo a posto, corriamo, di nuovo dalla console:

```
mkdir our_project
cd our_project
```

Ora siamo nel posto in cui il nostro codice vivrà. Per creare l'archivio principale del nostro progetto puoi eseguire

Ok, ma come creiamo il progetto express skeleton?

È semplice:

```
npm install -g express express-generator
```

Le distribuzioni Linux e Mac dovrebbero usare **sudo** per installarlo perché sono installate nella directory nodejs che è accessibile solo dall'utente **root** . Se tutto è andato bene possiamo finalmente creare lo scheletro Express-App, basta correre

```
express
```

Questo comando creerà all'interno della nostra cartella un'app di esempio esplicita. La struttura è la seguente:

```
bin/
public/
routes/
views/
app.js
package.json
```

Ora, se eseguiamo **npm, iniziamo a** visitare http: // localhost: 3000 vedremo l'applicazione Express funzionante, abbastanza bene abbiamo generato un'app express senza troppi problemi, ma come possiamo mescolarlo con AngularJS? .

Come funziona espresso, brevemente?

Express è un framework basato su **Nodejs** , puoi vedere la documentazione ufficiale sul sito Express . Ma per il nostro scopo abbiamo bisogno di sapere che **Express** è il responsabile quando scriviamo, ad esempio, http: // localhost: 3000 / home del rendering della home page della nostra applicazione. Dall'app creata di recente, possiamo verificare:

```
FILE: routes/index.js
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

Quello che questo codice ci sta dicendo è che quando l'utente va su http: // localhost: 3000 deve rendere la vista **indice** e passare un **JSON** con una proprietà title e valore Express. Ma quando controlliamo la directory views e apriamo index.jade possiamo vedere questo:

```
extends layout
block content
    h1= title
    p Welcome to #{title}
```

Questa è un'altra potente funzionalità Express, i **motori di template** , che ti permettono di eseguire il rendering del contenuto nella pagina passando variabili ad esso o ereditare un altro modello in modo che le tue pagine siano più compatte e meglio comprensibili da altri. L'estensione del file è **.jade** per quanto ne so **Jade ha** cambiato il nome per **Pug** , fondamentalmente è lo stesso motore di template ma con alcuni aggiornamenti e modifiche fondamentali.

Installazione di Pug e aggiornamento del motore di template Express.

Ok, per iniziare a usare Pug come motore di template del nostro progetto dobbiamo eseguire:

```
npm install --save pug
```

Questo installerà Pug come dipendenza del nostro progetto e lo salverà su **package.json** . Per usarlo abbiamo bisogno di modificare il file **app.js** :

```
var app = express();
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
```

E sostituisci il motore della linea di vista con Carlino e basta. Possiamo avviare nuovamente il nostro progetto con **l'avvio di npm** e vedremo che tutto funziona **correttamente** .

In che modo AngularJS si inserisce in tutto questo?

AngularJS è un **framework MVW** Javascript (Model-View-Whatever) utilizzato principalmente per creare l'installazione di **SPA** (Simple Page Application) è abbastanza semplice, è possibile visitare il sito Web di AngularJS e scaricare l'ultima versione **v1.6.4** .

Dopo aver scaricato AngularJS quando dovremmo copiare il file nella nostra cartella **public / javascripts** all'interno del nostro progetto, una piccola spiegazione, questa è la cartella che serve le risorse statiche del nostro sito, immagini, css, file javacript e così via. Naturalmente questo è configurabile tramite il file **app.js** , ma lo manterremo semplice. Ora creiamo un file chiamato **ng-app.js** , il file in cui la nostra applicazione vivrà, all'interno della nostra cartella pubblica dei javascript, proprio dove vive AngularJS. Per portare su AngularJS, è necessario modificare il contenuto delle **viste / layout.pug** come segue:

```
doctype html
html(ng-app='first-app')
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body(ng-controller='indexController')
    block content

  script(type='text-javascript', src='javascripts/angular.min.js')
  script(type='text-javascript', src='javascripts/ng-app.js')
```

Cosa stiamo facendo qui ?, beh, stiamo includendo il core di AngularJS e il nostro file **ng-app.js di** recente creazione **,** quindi quando il template è renderizzato porterà AngularJS in su, noterete l'uso della direttiva **ng-app** , questo è indicativo AngularJS che questo è il nostro nome dell'applicazione e dovrebbe attenersi ad esso.
Quindi, il contenuto del nostro **ng-app.js** sarà:

```
angular.module('first-app', [])
       .controller('indexController', ['$scope', indexController]);

function indexController($scope) {
    $scope.name = 'sigfried';
}
```

Stiamo utilizzando la funzione AngularJS più basilare qui, **associazione dati bidirezionale** , questo ci permette di aggiornare istantaneamente il contenuto della nostra vista e controller, questa è una spiegazione molto semplice, ma puoi fare una ricerca in Google o StackOverflow per vedere come funziona davvero.

Quindi, abbiamo i blocchi di base della nostra applicazione AngularJS, ma c'è qualcosa che dobbiamo fare, dobbiamo aggiornare la nostra pagina index.pug per vedere i cambiamenti della nostra app angolare, facciamolo:

```
extends layout
block content
  div(ng-controller='indexController')
    h1= title
    p Welcome {{name}}
    input(type='text' ng-model='name')
```

Qui leghiamo semplicemente l'input al nostro nome di proprietà definito nell'oscilloscopio AngularJS all'interno del nostro controller:

```
$scope.name = 'sigfried';
```

Lo scopo di questo è che ogni volta che cambiamo il testo nell'input il paragrafo precedente aggiornerà il suo contenuto all'interno del {{nome}}, questo è chiamato **interpolazione** , un'altra caratteristica AngularJS per rendere il nostro contenuto nel modello.

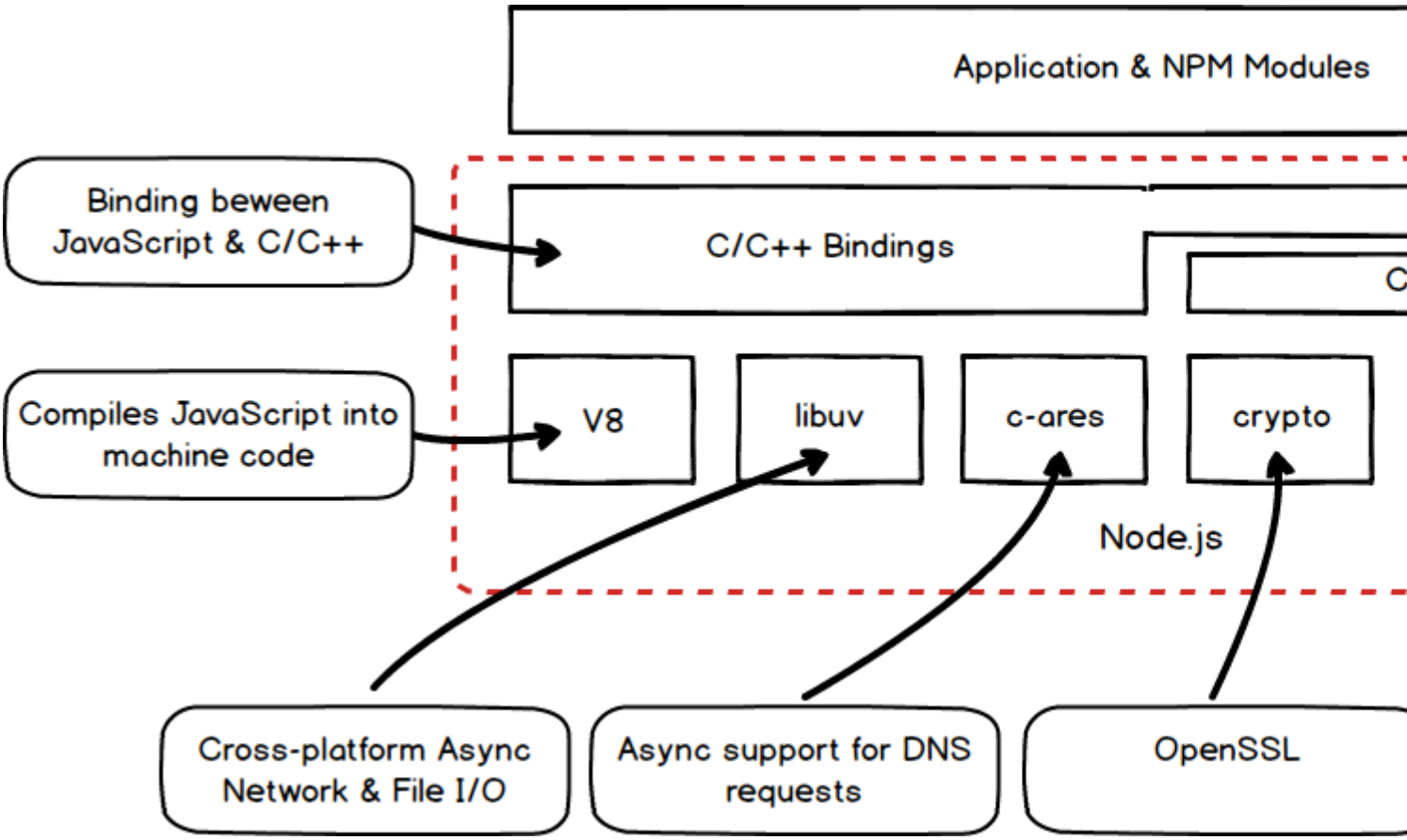
Quindi, tutto è configurato, ora possiamo eseguire **npm,** andare su http: // localhost: 3000 e vedere la nostra applicazione Express che serve la pagina e AngularJS che gestisce il frontend dell'applicazione.

Leggi Node.js (express.js) con il codice di esempio angular.js online: https://riptutorial.com/it/node-js/topic/9757/node-js--express-js--con-il-codice-di-esempio-angular-js

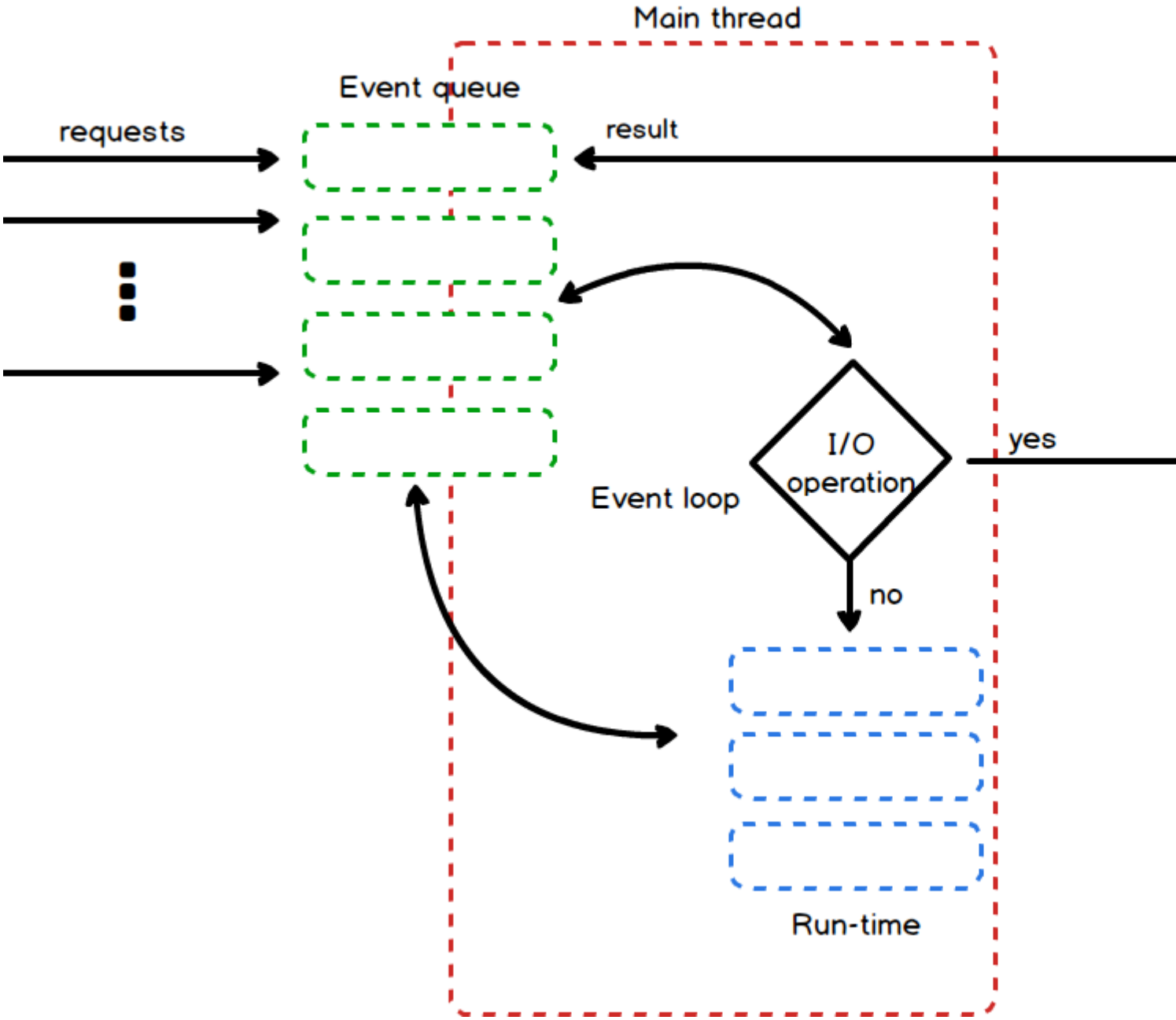
Capitolo 74: Node.js Architecture & Inner Workings

Examples

Node.js - sotto il cofano



Node.js - in movimento



Leggi Node.js Architecture & Inner Workings online: https://riptutorial.com/it/node-js/topic/5892/node-js-architecture--amp--inner-workings

Capitolo 75: Node.js con CORS

Examples

Abilita CORS in express.js

Poiché node.js viene spesso utilizzato per creare API, l'impostazione CORS corretta può essere un salvataggio se si desidera poter richiedere l'API da domini diversi.

Nell'esempio, lo configureremo per la configurazione più ampia (autorizza tutti i tipi di richieste da qualsiasi dominio.

Nel server.js dopo l'inizializzazione di express:

```
// Create express server
const app = express();

app.use((req, res, next) => {
    res.header('Access-Control-Allow-Origin', '*');

    // authorized headers for preflight requests
    // https://developer.mozilla.org/en-US/docs/Glossary/preflight_request
    res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type,
Accept');
    next();

    app.options('*', (req, res) => {
        // allowed XHR methods
        res.header('Access-Control-Allow-Methods', 'GET, PATCH, PUT, POST, DELETE, OPTIONS');
        res.send();
    });
});
```

Di solito, il nodo viene eseguito dietro un proxy sui server di produzione. Pertanto il server proxy inverso (come Apache o Nginx) sarà responsabile della configurazione di CORS.

Per adattare convenientemente questo scenario, è possibile abilitare solo node.js CORS quando è in fase di sviluppo.

Questo è fatto facilmente controllando `NODE_ENV` :

```
const app = express();

if (process.env.NODE_ENV === 'development') {
    // CORS settings
}
```

Leggi Node.js con CORS online: https://riptutorial.com/it/node-js/topic/9272/node-js-con-cors

Capitolo 76: Node.JS con ES6

introduzione

ES6, ECMAScript 6 o ES2015 è l'ultima specifica per JavaScript che introduce dello zucchero sintattico nella lingua. È un grande aggiornamento del linguaggio e introduce molte nuove funzionalità

Maggiori dettagli su Node e ES6 possono essere trovati sul loro sito https://nodejs.org/en/docs/es6/

Examples

Nodo ES6 Supporto e creazione di un progetto con Babel

L'intera specifica ES6 non è stata ancora implementata nella sua interezza, quindi sarà possibile utilizzare solo alcune delle nuove funzionalità. Puoi vedere un elenco delle funzionalità ES6 attualmente supportate su http://node.green/

Dal momento che NodeJS v6 ha avuto un supporto abbastanza buono. Pertanto, se si utilizza NodeJS v6 o versione successiva, è possibile utilizzare ES6. Tuttavia, potresti anche voler utilizzare alcune delle funzioni inedite e alcune da oltre. Per questo dovrai usare un transpiler

È possibile eseguire un transpiler in fase di esecuzione e creare, utilizzare tutte le funzionalità di ES6 e altro ancora. Il traspiatore più popolare per JavaScript si chiama Babel

Babel ti consente di utilizzare tutte le funzionalità delle specifiche ES6 e alcune funzionalità aggiuntive non specificate con "stage-0", ad esempio `import thing from 'thing` invece di `var thing = require('thing')`

Se volessimo creare un progetto in cui usassimo funzionalità 'stage-0' come l'importazione, avremmo bisogno di aggiungere Babel come traspiatore. Vedrai i progetti usando react e Vue e altri pattern basati su CommonJS implementeranno lo stage-0 abbastanza spesso.

crea un nuovo progetto di nodo

```
mkdir my-es6-app
cd my-es6-app
npm init
```

Installa babel sul preset ES6 e sullo stage 0

```
npm install --save-dev babel-preset-es2015 babel-preset-stage-2 babel-cli babel-register
```

Crea un nuovo file chiamato `server.js` e aggiungi un server HTTP di base.

```
import http from 'http'
```

```
http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'})
  res.end('Hello World\n')
}).listen(3000, '127.0.0.1')

console.log('Server running at http://127.0.0.1:3000/')
```

Nota che usiamo un `import http from 'http'` questa è una funzione di stage-0 e se funziona significa che abbiamo il transpiler che funziona correttamente.

Se si esegue `node server.js` non riuscirà a sapere come gestire l'importazione.

Creare un file .babelrc nella directory principale della directory e aggiungere le seguenti impostazioni

```
{
  "presets": ["es2015", "stage-2"],
  "plugins": []
}
```

ora puoi eseguire il server con il `node src/index.js --exec babel-node`

Completare l'operazione non è una buona idea eseguire un transpiler in fase di esecuzione su un'app di produzione. Tuttavia, possiamo implementare alcuni script nel nostro package.json per semplificare il lavoro.

```
"scripts": {
    "start": "node dist/index.js",
    "dev": "babel-node src/index.js",
    "build": "babel src -d dist",
    "postinstall": "npm run build"
  },
```

Quanto sopra sopra `npm install` il codice transpiled nella directory dist consentendo a `npm start` a usare il codice transpiled per la nostra app di produzione.

`npm run dev` avvierà il server e il runtime di babel che va bene e preferito quando si lavora su un progetto localmente.

Andando avanti, è possibile installare nodemon `npm install nodemon --save-dev` per controllare le modifiche e quindi riavviare l'app nodo.

Questo velocizza davvero il lavoro con Babel e NodeJS. In you package.json basta aggiornare lo script "dev" per usare nodemon

```
"dev": "nodemon src/index.js --exec babel-node",
```

Usa JS es6 sulla tua app NodeJS

JS es6 (noto anche come es2015) è un insieme di nuove funzionalità del linguaggio JS volte a renderlo più intuitivo quando si utilizza OOP o mentre si affrontano le attività di sviluppo moderne.

Prerequisiti:

1. Scopri le nuove funzionalità di es6 su http://es6-features.org - potrebbe chiarirti se hai intenzione di usarlo nella tua prossima app NodeJS
2. Verifica il livello di compatibilità della versione del tuo nodo su http://node.green
3. Se tutto va bene, facciamo il codice!

Ecco un esempio molto breve di una semplice app `hello world` con JS es6

```
'use strict'

class Program
{
    constructor()
    {
        this.message = 'hello es6 :)';
    }

    print()
    {
        setTimeout(() =>
        {
            console.log(this.message);

            this.print();

        }, Math.random() * 1000);
    }
}

new Program().print();
```

È possibile eseguire questo programma e osservare come stampa lo stesso messaggio più e più volte.

Ora .. lasciamolo giù riga per riga:

```
'use strict'
```

Questa linea è effettivamente necessaria se si intende utilizzare js es6. `strict` modalità `strict` , intenzionalmente, ha una semantica diversa dal codice normale (si prega di leggere di più su MDN - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)

```
class Program
```

Incredibile: una parola chiave di `class` ! Solo per un rapido riferimento - prima di es6 l'unico modo per definire una classe in js era con la parola chiave ... `function` !

```
function MyClass() // class definition
```

```
{

}

var myClassObject = new MyClass(); // generating a new object with a type of MyClass
```

Quando si utilizza OOP, una classe è un'abilità fondamentale che aiuta lo sviluppatore a rappresentare una parte specifica di un sistema (la decomposizione del codice è cruciale quando il codice diventa più grande .. ad esempio: quando si scrive il codice lato server)

```
constructor()
{
    this.message = 'hello es6 :)';
}
```

Devi ammettere che è abbastanza intuitivo! Questo è l'indice della mia classe - questa "funzione" unica si verificherà ogni volta che un oggetto viene creato da questa particolare classe (nel nostro programma - solo una volta)

```
print()
{
    setTimeout(() => // this is an 'arrow' function
    {
        console.log(this.message);

        this.print(); // here we call the 'print' method from the class template itself (a
recursion in this particular case)

    }, Math.random() * 1000);
}
```

Perché la stampa è definita nell'ambito della classe - in realtà è un metodo - che può essere invocato dall'oggetto della classe o dalla stessa classe!

Quindi .. fino ad ora abbiamo definito la nostra classe .. tempo di usarla:

```
new Program().print();
```

Che è veramente uguale a:

```
var prog = new Program(); // define a new object of type 'Program'

prog.print(); // use the program to print itself
```

In conclusione: JS es6 può semplificare il tuo codice - renderlo più intuitivo e facile da capire (confrontandolo con la versione precedente di JS) .. potresti provare a riscrivere un tuo codice esistente e vedere la differenza per te stesso

GODERE :)

Leggi Node.JS con ES6 online: https://riptutorial.com/it/node-js/topic/5934/node-js-con-es6

Capitolo 77: Node.js con Oracle

Examples

Connetti a Oracle DB

Un modo molto semplice per connettersi a un database ORACLE è utilizzando il modulo `oracledb` . Questo modulo gestisce la connessione tra l'app Node.js e il server Oracle. Puoi installarlo come qualsiasi altro modulo:

```
npm install oracledb
```

Ora devi creare una connessione ORACLE, che puoi in seguito interrogare.

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

oracledb.getConnection(
  {
    user          : "oli",
    password      : "password",
    connectString : "ORACLE_DEV_DB_TNS_NAME"
  },
  connExecute
);
```

Il connectString "ORACLE_DEV_DB_TNA_NAME" può vivere in un file tnsnames.org nella stessa directory o in cui è installato il tuo client istantaneo Oracle.

Se non si dispone di alcun client istantaneo Oracle installato sul proprio computer di sviluppo, è possibile seguire la `instant client installation guide` per il proprio sistema operativo.

Interrogare un oggetto di connessione senza parametri

Utilizzare ora può utilizzare la funzione connExecute per l'esecuzione di una query. Hai la possibilità di ottenere il risultato della query come oggetto o array. Il risultato è stampato su console.log.

```
function connExecute(err, connection)
{
    if (err) {
        console.error(err.message);
        return;
    }
    sql = "select 'test' as c1, 'oracle' as c2 from dual";
    connection.execute(sql, {}, { outFormat: oracledb.OBJECT }, // or oracledb.ARRAY
        function(err, result)
        {
            if (err) {
                console.error(err.message);
                connRelease(connection);
```

```
                return;
            }
            console.log(result.metaData);
            console.log(result.rows);
            connRelease(connection);
        });
}
```

Poiché abbiamo utilizzato una connessione non di pooling, dobbiamo rilasciare nuovamente la nostra connessione.

```
function connRelease(connection)
{
  connection.close(
    function(err) {
      if (err) {
        console.error(err.message);
      }
    });
}
```

L'output per un oggetto sarà

```
[ { name: 'C1' }, { name: 'C2' } ]
[ { C1: 'test', C2: 'oracle' } ]
```

e l'output per un array sarà

```
[ { name: 'C1' }, { name: 'C2' } ]
[ [ 'test', 'oracle' ] ]
```

Utilizzo di un modulo locale per interrogazioni più semplici

Per semplificare le tue query da ORACLE-DB, puoi chiamare la tua query in questo modo:

```
const oracle = require('./oracle.js');

const sql = "select 'test' as c1, 'oracle' as c2 from dual";
oracle.queryObject(sql, {}, {})
    .then(function(result) {
        console.log(result.rows[0]['C2']);
    })
    .catch(function(err) {
        next(err);
    });
```

La creazione della connessione e l'esecuzione sono inclusi in questo file oracle.js con il contenuto come segue:

```
'use strict';
const oracledb = require('oracledb');

const oracleDbRelease = function(conn) {
```

```
  conn.release(function (err) {
    if (err)
      console.log(err.message);
  });
};

function queryArray(sql, bindParams, options) {
    options.isAutoCommit = false; // we only do SELECTs

    return new Promise(function(resolve, reject) {
        oracledb.getConnection(
                  {
                    user          : "oli",
                    password      : "password",
                    connectString : "ORACLE_DEV_DB_TNA_NAME"
                })
        .then(function(connection){
            //console.log("sql log: " + sql + " params " + bindParams);
            connection.execute(sql, bindParams, options)
            .then(function(results) {
                resolve(results);
                process.nextTick(function() {
                    oracleDbRelease(connection);
                });
            })
            .catch(function(err) {
                reject(err);

                process.nextTick(function() {
                    oracleDbRelease(connection);
                        });
                    });
            })
            .catch(function(err) {
                reject(err);
            });
    });
}

function queryObject(sql, bindParams, options) {
    options['outFormat'] = oracledb.OBJECT; // default is oracledb.ARRAY
    return queryArray(sql, bindParams, options);
}

module.exports = queryArray;
module.exports.queryArray = queryArray;
module.exports.queryObject = queryObject;
```

Nota che hai entrambi i metodi queryArray e queryObject per chiamare sul tuo oggetto oracle.

Leggi Node.js con Oracle online: https://riptutorial.com/it/node-js/topic/8248/node-js-con-oracle

Capitolo 78: Node.js Design Fundamental

Examples

La filosofia Node.js

Small Core , **Small Module** : -

Costruisci moduli per piccoli e singoli scopi non solo in termini di dimensioni del codice, ma anche in termini di ambito che serve a un unico scopo

```
    a - "Small is beautiful"
    b - "Make each program do one thing well."
```

Il modello del reattore

The Reactor Pattern è il cuore della natura asincrona di `node.js` Consentito al sistema di essere implementato come un processo a thread singolo con una serie di generatori di eventi e gestori di eventi, con l'aiuto del ciclo di eventi che viene eseguito continuamente.

Il motore I / O non bloccante di Node.js - libuv -

The Observer Pattern (EventEmitter) mantiene un elenco di dipendenti / osservatori e li notifica

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

var ringBell = function ringBell()
{
  console.log('tring tring tring');
}
eventEmitter.on('doorOpen', ringBell);

eventEmitter.emit('doorOpen');
```

Leggi Node.js Design Fundamental online: https://riptutorial.com/it/node-js/topic/6274/node-js-design-fundamental

Capitolo 79: Node.JS e MongoDB.

Osservazioni

Queste sono le operazioni CRUD di base per l'uso di mongo db con nodejs.

Domanda: ci sono altri modi in cui puoi fare ciò che viene fatto qui ??

Risposta: Sì, ci sono numerosi modi per farlo.

Domanda: Sta usando la mangusta ??

Risposta: No. Ci sono altri pacchetti disponibili che possono aiutarti.

Domanda: dove posso ottenere la documentazione completa di mangusta ??

Risposta: clicca qui

Examples

Connessione a un database

Per connettersi a un database mongo dall'applicazione nodo richiediamo mangusta.

Installazione di Mongoose Vai al toot della tua applicazione e installa mangusta di

```
npm install mongoose
```

Quindi ci colleghiamo al database.

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

//connect to the test database running on default mongod port of localhost
mongoose.connect('mongodb://localhost/test');



//Connecting with custom credentials
mongoose.connect('mongodb://USER:PASSWORD@HOST:PORT/DATABASE');


//Using Pool Size to define the number of connections opening
//Also you can use a call back function for error handling
mongoose.connect('mongodb://localhost:27017/consumers',
                 {server: { poolSize: 50 }},
                 function(err) {
                    if(err) {
                        console.log('error in this')
                        console.log(err);
                        // Do whatever to handle the error
                    } else {
```

```
                        console.log('Connected to the database');
                    }
                });
```

Creazione di una nuova collezione

Con Mongoose, tutto è derivato da uno schema. Consente di creare uno schema.

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

var Schema = mongoose.Schema;

var AutoSchema = new Schema({
    name : String,
    countOf: Number,
});
// defining the document structure

// by default the collection created in the db would be the first parameter we use (or the
plural of it)
module.exports = mongoose.model('Auto', AutoSchema);

// we can over write it and define the collection name by specifying that in the third
parameters.
module.exports = mongoose.model('Auto', AutoSchema, 'collectionName');


// We can also define methods in the models.
AutoSchema.methods.speak = function () {
  var greeting = this.name
    ? "Hello this is  " + this.name+ " and I have counts of "+ this.countOf
    : "I don't have a name";
  console.log(greeting);
}
mongoose.model('Auto', AutoSchema, 'collectionName');
```

Ricorda che i metodi devono essere aggiunti allo schema prima di compilarlo con mongoose.model () come fatto sopra.

Inserimento di documenti

Per inserire un nuovo documento nella collezione, creiamo un oggetto dello schema.

```
var Auto = require('models/auto')
var autoObj = new Auto({
    name: "NewName",
    countOf: 10
});
```

Lo salviamo come il seguente

```
autoObj.save(function(err, insertedAuto) {
    if (err) return console.error(err);
    insertedAuto.speak();
    // output: Hello this is NewName and I have counts of 10
```

```
});
```

Questo inserirà un nuovo documento nella collezione

Lettura

Leggere i dati dalla collezione è molto semplice. Ottenere tutti i dati della collezione.

```
var Auto = require('models/auto')
Auto.find({}, function (err, autos) {
      if (err) return console.error(err);
       // will return a json array of all the documents in the collection
      console.log(autos);
})
```

Lettura dei dati con una condizione

```
Auto.find({countOf: {$gte: 5}}, function (err, autos) {
      if (err) return console.error(err);
       // will return a json array of all the documents in the collection whose count is
greater than 5
      console.log(autos);
})
```

È inoltre possibile specificare il secondo parametro come oggetto di ciò che tutti i campi necessari

```
Auto.find({},{name:1}, function (err, autos) {
      if (err) return console.error(err);
       // will return a json array of name field of all the documents in the collection
      console.log(autos);
})
```

Trovare un documento in una raccolta.

```
Auto.findOne({name:"newName"}, function (err, auto) {
      if (err) return console.error(err);
     //will return the first object of the document whose name is "newName"
      console.log(auto);
})
```

Trovare un documento in una raccolta tramite id.

```
Auto.findById(123, function (err, auto) {
      if (err) return console.error(err);
     //will return the first json object of the document whose id is 123
      console.log(auto);
})
```

In aggiornamento

Per l'aggiornamento di raccolte e documenti possiamo utilizzare uno di questi metodi:

metodi

- aggiornare()
- updateOne ()
- updateMany ()
- replaceOne ()

Aggiornare()

Il metodo update () modifica uno o più documenti (parametri di aggiornamento)

```
db.lights.update(
   { room: "Bedroom" },
   { status: "On" }
)
```

Questa operazione cerca la raccolta 'luci' per un documento in cui `room` è **Bedroom** *(1 ° parametro)* . Quindi aggiorna la proprietà dello `status` documenti corrispondenti su **On** *(2 ° parametro)* e restituisce un oggetto WriteResult simile al seguente:

```
{ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }
```

UpdateOne

Il metodo UpdateOne () modifica UN documento (parametri di aggiornamento)

```
db.countries.update(
   { country: "Sweden" },
   { capital: "Stockholm" }
)
```

Questa operazione cerca nella raccolta "Paesi" un documento in cui il `country` è la **Svezia** *(1 ° parametro)* . Quindi aggiorna la `capital` proprietà dei documenti corrispondenti a **Stoccolma** *(2 ° parametro)* e restituisce un oggetto WriteResult simile al seguente:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

UpdateMany

Il metodo UpdateMany () modifica i documenti multible (parametri di aggiornamento)

```
db.food.updateMany(
   { sold: { $lt: 10 } },
   { $set: { sold: 55 } }
)
```

Questa operazione aggiorna tutti i documenti *(in una raccolta "cibo")* dove la `sold` è **inferiore a 10** * (1 ° parametro) impostando `sold` a **55** . Quindi restituisce un oggetto WriteResult simile al seguente:

```
{ "acknowledged" : true, "matchedCount" : a, "modifiedCount" : b }
```

a = Numero di documenti corrispondenti
b = numero di documenti modificati

ReplaceOne

Sostituisce il primo documento di corrispondenza (documento sostitutivo)

Questo esempio di raccolta denominata **Paesi** contiene 3 documenti:

```
{ "_id" : 1, "country" : "Sweden" }
{ "_id" : 2, "country" : "Norway" }
{ "_id" : 3, "country" : "Spain" }
```

La seguente operazione sostituisce il documento `{ country: "Spain" }` con documento `{ country: "Finland" }`

```
db.countries.replaceOne(
   { country: "Spain" },
   { country: "Finland" }
)
```

E ritorna:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

I **Paesi di** raccolta di esempio ora contengono:

```
{ "_id" : 1, "country" : "Sweden" }
{ "_id" : 2, "country" : "Norway" }
{ "_id" : 3, "country" : "Finland" }
```

Eliminazione

L'eliminazione di documenti da una raccolta in mangusta avviene nel modo seguente.

```
Auto.remove({_id:123}, function(err, result){
    if (err) return console.error(err);
    console.log(result); // this will specify the mongo default delete result.
});
```

Leggi Node.JS e MongoDB. online: https://riptutorial.com/it/node-js/topic/7505/node-js-e-mongodb-

Capitolo 80: Node.js v6 Nuove funzionalità e miglioramenti

introduzione

Con il nodo 6 diventa la nuova versione LTS del nodo. Possiamo vedere una serie di miglioramenti alla lingua attraverso i nuovi standard ES6. Passeremo attraverso alcune delle nuove funzionalità introdotte e alcuni esempi su come implementarle.

Examples

Parametri funzione predefiniti

```
function addTwo(a, b = 2) {
    return a + b;
}

addTwo(3) // Returns the result 5
```

Con l'aggiunta di parametri di funzione predefiniti ora puoi rendere opzionali gli argomenti e impostarli come predefiniti su un valore a tua scelta.

Parametri di riposo

```
function argumentLength(...args) {
    return args.length;
}

argumentLength(5) // returns 1
argumentLength(5, 3) //returns 2
argumentLength(5, 3, 6) //returns 3
```

Prefacendo l'ultimo argomento della tua funzione con `...` tutti gli argomenti passati alla funzione vengono letti come una matrice. In questo esempio otteniamo pass in più argomenti e otteniamo la lunghezza della matrice creata da quegli argomenti.

Spread Operator

```
function myFunction(x, y, z) { }
var args = [0, 1, 2];
myFunction(...args);
```

La sintassi di diffusione consente di espandere un'espressione in luoghi in cui sono previsti più argomenti (per chiamate di funzione) o più elementi (per letterali di array) o più variabili. Proprio come i parametri di riposo, semplicemente prefigurano il tuo array con `...`

Funzioni della freccia

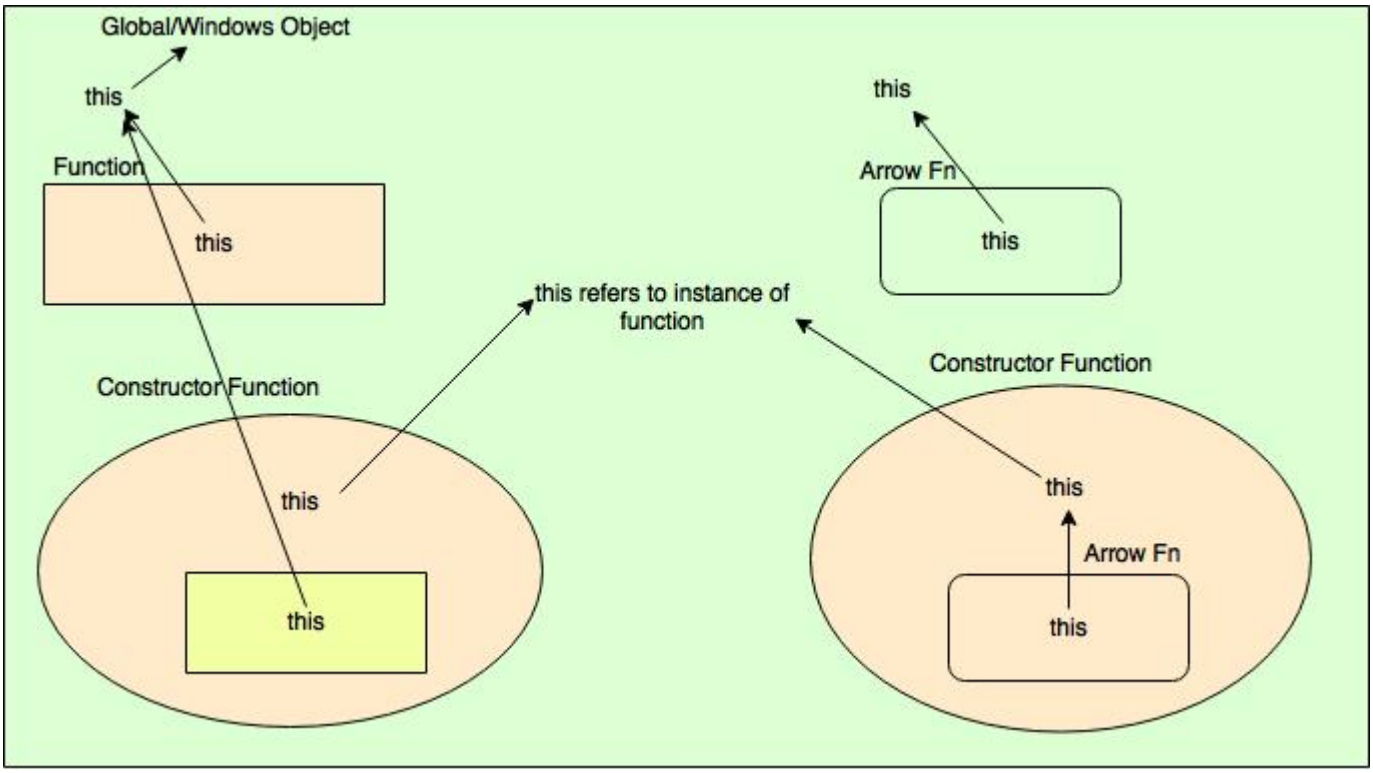
La funzione Arrow è il nuovo modo di definire una funzione in ECMAScript 6.

```
// traditional way of declaring and defining function
var sum = function(a,b)
{
    return a+b;
}

// Arrow Function
let sum = (a, b)=> a+b;

//Function defination using multiple lines
let checkIfEven = (a) => {
    if( a % 2 == 0 )
        return true;
    else
        return false;
}
```

"questo" in Arrow Function

questa funzione si riferisce ad oggetto istanza utilizzato per chiamare tale funzione ma ***questa*** funzione freccia è uguale a *questo* di funzione in cui è definita la funzione freccia.

Capiamo usando il diagramma



Comprensione con esempi.

```
var normalFn = function(){
   console.log(this) // refers to global/window object.
```

```
}

var arrowFn = () => console.log(this); // refers to window or global object as function is
defined in scope of global/window object

var service = {

    constructorFn : function(){

        console.log(this); //  refers to service as service object used to call method.

        var nestedFn = function(){
            console.log(this); // refers window or global object because no instance object
was used to call this method.
        }
        nestedFn();
    },

    arrowFn : function(){
        console.log(this); // refers to service as service object was used to call method.
        let fn = () => console.log(this); // refers to service object as arrow function
defined in function which is called using instance object.
        fn();
    }
}

// calling defined functions
constructorFn();
arrowFn();
service.constructorFn();
service.arrowFn();
```

Nella funzione freccia, *questo* è lo scope lessicale che è l'ambito della funzione in cui è definita la funzione freccia.
Il primo esempio è il modo tradizionale di definire le funzioni e quindi *si* riferisce all'oggetto *globale / finestra* .
Nel secondo esempio *questo* è usato all'interno della funzione freccia, quindi *questo si* riferisce all'ambito in cui è definito (che è Windows o oggetto globale). Nel terzo esempio *si* tratta di oggetto di servizio poiché l'oggetto di servizio viene utilizzato per chiamare la funzione.
Nel quarto esempio, la funzione freccia è definita e chiamata dalla funzione il cui ambito è *servizio* , quindi stampa oggetto *servizio* .

Nota: - l'oggetto globale viene stampato in Node.Js e l'oggetto Windows nel browser.

Leggi Node.js v6 Nuove funzionalità e miglioramenti online: https://riptutorial.com/it/node-js/topic/8593/node-js-v6-nuove-funzionalita-e-miglioramenti

Capitolo 81: NodeJS con Redis

Osservazioni

Abbiamo coperto le operazioni di base e più comunemente utilizzate in node_redis. Puoi utilizzare questo modulo per sfruttare tutta la potenza di Redis e creare app Node.js davvero sofisticate. Puoi creare molte cose interessanti con questa libreria come un forte livello di memorizzazione nella cache, un potente sistema di messaggistica Pub / Sub e molto altro. Per saperne di più sulla biblioteca, consultare la documentazione .

Examples

Iniziare

node_redis, come avrai intuito, è il client Redis per Node.js. È possibile installarlo tramite npm utilizzando il seguente comando.

```
npm install redis
```

Una volta installato il modulo node_redis, sei a posto. Creiamo un semplice file, app.js, e vediamo come connettersi con Redis da Node.js.

app.js

```
var redis = require('redis');
client = redis.createClient(); //creates a new client
```

Per impostazione predefinita, redis.createClient () utilizzerà 127.0.0.1 e 6379 rispettivamente come nome host e porta. Se hai un host / porta diverso, puoi fornirli come segue:

```
var client = redis.createClient(port, host);
```

Ora puoi eseguire qualche azione una volta stabilita una connessione. Fondamentalmente, devi solo ascoltare gli eventi di connessione come mostrato di seguito.

```
client.on('connect', function() {
    console.log('connected');
});
```

Quindi, il seguente snippet va in app.js:

```
var redis = require('redis');
var client = redis.createClient();

client.on('connect', function() {
    console.log('connected');
```

```
});
```

Ora, digitare l'app nodo nel terminale per eseguire l'app. Assicurati che il tuo server Redis sia attivo e funzionante prima di eseguire questo snippet.

Memorizzare coppie chiave-valore

Ora che sai come connetterti con Redis da Node.js, vediamo come memorizzare coppie chiave-valore nell'archivio Redis.

Memorizzazione di stringhe

Tutti i comandi di Redis sono esposti come funzioni differenti sull'oggetto client. Per memorizzare una stringa semplice usa la seguente sintassi:

```
client.set('framework', 'AngularJS');
```

O

```
client.set(['framework', 'AngularJS']);
```

I frammenti di cui sopra memorizzano una semplice stringa AngularJS rispetto al framework chiave. Dovresti notare che entrambi i frammenti fanno la stessa cosa. L'unica differenza è che il primo passa un numero variabile di argomenti mentre il successivo passa un array di argomenti alla funzione `client.set()` . È inoltre possibile passare una richiamata opzionale per ricevere una notifica quando l'operazione è completata:

```
client.set('framework', 'AngularJS', function(err, reply) {
  console.log(reply);
});
```

Se l'operazione non è riuscita per qualche motivo, l'argomento `err` del callback rappresenta l'errore. Per recuperare il valore della chiave, procedi come segue:

```
client.get('framework', function(err, reply) {
    console.log(reply);
});
```

`client.get()` consente di recuperare una chiave memorizzata in Redis. È possibile accedere al valore della chiave tramite la risposta dell'argomento di richiamata. Se la chiave non esiste, il valore della risposta sarà vuoto.

Memorizzazione di hash

Molte volte la memorizzazione di valori semplici non risolverà il tuo problema. Sarà necessario memorizzare gli hash (oggetti) in Redis. Per questo è possibile utilizzare la funzione `hmset()` come segue:

```
client.hmset('frameworks', 'javascript', 'AngularJS', 'css', 'Bootstrap', 'node', 'Express');

client.hgetall('frameworks', function(err, object) {
    console.log(object);
});
```

Il frammento di cui sopra memorizza un hash in Redis che associa ogni tecnologia al suo framework. Il primo argomento su `hmset()` è il nome della chiave. Gli argomenti successivi rappresentano coppie chiave-valore. Allo stesso modo, `hgetall()` viene utilizzato per recuperare il valore della chiave. Se viene trovata la chiave, il secondo argomento del callback conterrà il valore che è un oggetto.

Si noti che Redis non supporta oggetti nidificati. Tutti i valori delle proprietà nell'oggetto verranno convertiti in stringhe prima di essere archiviati. È anche possibile utilizzare la seguente sintassi per memorizzare oggetti in Redis:

```
client.hmset('frameworks', {
    'javascript': 'AngularJS',
    'css': 'Bootstrap',
    'node': 'Express'
});
```

È anche possibile passare una richiamata opzionale per sapere quando l'operazione è completata.

Tutte le funzioni (comandi) possono essere richiamate con equivalenti maiuscoli / minuscoli. Ad esempio, `client.hmset()` e `client.HMSET()` sono gli stessi. Memorizzazione delle liste

Se si desidera memorizzare un elenco di elementi, è possibile utilizzare gli elenchi di Redis. Per memorizzare un elenco, utilizzare la seguente sintassi:

```
client.rpush(['frameworks', 'angularjs', 'backbone'], function(err, reply) {
    console.log(reply); //prints 2
});
```

Lo snippet sopra riportato crea un elenco chiamato framework e ne spinge due elementi. Quindi, la lunghezza della lista ora è due. Come puoi vedere ho passato un array di `args` a `rpush` . Il primo elemento della matrice rappresenta il nome della chiave mentre il resto rappresenta gli elementi della lista. Puoi anche usare `lpush()` invece di `rpush()` per spingere gli elementi a sinistra.

Per recuperare gli elementi della lista puoi usare la funzione `lrange()` come segue:

```
client.lrange('frameworks', 0, -1, function(err, reply) {
    console.log(reply); // ['angularjs', 'backbone']
});
```

Basta notare che si ottengono tutti gli elementi della lista passando -1 come terzo argomento di `lrange()` . Se vuoi un sottoinsieme della lista, dovresti passare l'indice finale qui.

Memorizzare i set

I set sono simili agli elenchi, ma la differenza è che non consentono duplicati. Quindi, se non vuoi elementi duplicati nella tua lista puoi usare un set. Ecco come possiamo modificare il nostro snippet precedente per utilizzare un set anziché un elenco.

```
client.sadd(['tags', 'angularjs', 'backbonejs', 'emberjs'], function(err, reply) {
    console.log(reply); // 3
});
```

Come puoi vedere, la funzione `sadd()` crea un nuovo set con gli elementi specificati. Qui, la lunghezza del set è tre. Per recuperare i membri del set, utilizzare la funzione `smembers()` come segue:

```
client.smembers('tags', function(err, reply) {
    console.log(reply);
});
```

Questo snippet recupererà tutti i membri del set. Basta notare che l'ordine non viene conservato durante il recupero dei membri.

Questa era una lista delle strutture dati più importanti trovate in ogni app alimentata da Redis. Oltre a stringhe, elenchi, insiemi e hash, è possibile memorizzare set ordinati, hyperLogLogs e altri in Redis. Se si desidera un elenco completo di comandi e strutture dati, visitare la documentazione ufficiale Redis. Ricorda che quasi ogni comando Redis è esposto sull'oggetto client offerto dal modulo node_redis.

Alcune operazioni più importanti supportate da node_redis.

Verifica dell'esistenza delle chiavi

A volte potrebbe essere necessario verificare se esiste già una chiave e procedere di conseguenza. Per fare ciò è possibile utilizzare la funzione `exists()` come mostrato di seguito:

```
client.exists('key', function(err, reply) {
    if (reply === 1) {
        console.log('exists');
    } else {
        console.log('doesn\'t exist');
    }
});
```

Cancellazione e scadenza delle chiavi

A volte è necessario cancellare alcune chiavi e reinizializzarle. Per cancellare i tasti, puoi usare il comando del comando come mostrato di seguito:

```
client.del('frameworks', function(err, reply) {
    console.log(reply);
});
```

Puoi anche dare un tempo di scadenza a una chiave esistente come segue:

```
client.set('key1', 'val1');
client.expire('key1', 30);
```

Lo snippet sopra riportato assegna un tempo di scadenza di 30 secondi alla chiave key1.

Incremento e decremento

Redis supporta anche le chiavi di incremento e decremento. Per incrementare una chiave, utilizzare la funzione `incr()` come mostrato di seguito:

```
client.set('key1', 10, function() {
    client.incr('key1', function(err, reply) {
        console.log(reply); // 11
    });
});
```

La funzione `incr()` incrementa un valore chiave di 1. Se è necessario incrementare di una quantità diversa, è possibile utilizzare la funzione `incrby()` . Allo stesso modo, per decrementare un tasto è possibile utilizzare le funzioni come `decr()` e `decrby()` .

Leggi NodeJS con Redis online: https://riptutorial.com/it/node-js/topic/7107/nodejs-con-redis

Capitolo 82: NodeJS Frameworks

Examples

Framework di server Web

Esprimere

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

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

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

Koa

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

app.use(function *(next){
  var start = new Date;
  yield next;
  var ms = new Date - start;
  console.log('%s %s - %s', this.method, this.url, ms);
});

app.use(function *(){
  this.body = 'Hello World';
});

app.listen(3000);
```

Command Line Interface Frameworks

Commander.js

```
var program = require('commander');

program
  .version('0.0.1')

program
  .command('hi')
  .description('initialize project configuration')
  .action(function(){
```

```
        console.log('Hi my Friend!!!');
});

program
  .command('bye [name]')
  .description('initialize project configuration')
  .action(function(name){
        console.log('Bye ' + name + '. It was good to see you!');
});

program
  .command('*')
  .action(function(env){
    console.log('Enter a Valid command');
    terminate(true);
});

program.parse(process.argv);
```

Vorpal.js

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

vorpal
  .command('foo', 'Outputs "bar".')
  .action(function(args, callback) {
    this.log('bar');
    callback();
  });

vorpal
  .delimiter('myapp$')
  .show();
```

Leggi NodeJS Frameworks online: https://riptutorial.com/it/node-js/topic/6042/nodejs-frameworks

Capitolo 83: npm

introduzione

Node Package Manager (npm) fornisce le seguenti due funzionalità principali: repository online per pacchetti / moduli node.js che sono ricercabili su search.nodejs.org. Utilità della riga di comando per installare i pacchetti Node.js, eseguire la gestione delle versioni e la gestione delle dipendenze dei pacchetti Node.js.

Sintassi

- npm <comando> dove <comando> è uno di:
  - Aggiungi utente
  - Aggiungi utente
  - apihelp
  - autore
  - bidone
  - bug
  - c
  - nascondiglio
  - completamento
  - config
  - DDP
  - deduplicazione
  - disapprovare
  - docs
  - modificare
  - Esplorare
  - FAQ
  - trova
  - trovare-gonzi
  - ottenere
  - Aiuto
  - help-search
  - casa
  - io
  - installare
  - Informazioni
  - dentro
  - istallare
  - problemi
  - la
  - collegamento
  - elenco

- ll
- ln
- accesso
- ls
- antiquato
- proprietario
- pacco
- prefisso
- fesso
- pubblicare
- r
- rb
- ricostruire
- rimuovere
- pronti contro termine
- ricomincia
- rm
- radice
- eseguire script
- S
- SE
- ricerca
- impostato
- mostrare
- shrinkwrap
- stella
- stelle
- inizio
- Stop
- modulo
- etichetta
- test
- tst
- un
- disinstallazione
- scollegare
- non pubblicato
- Rimuovi da speciali
- su
- aggiornare
- v
- versione
- vista
- chi sono

Parametri

Parametro	Esempio
accesso	`npm publish --access=public`
bidone	`npm bin -g`
modificare	`npm edit connect`
Aiuto	`npm help init`
dentro	`npm init`
installare	`npm install`
collegamento	`npm link`
fesso	`npm prune`
pubblicare	`npm publish ./`
ricomincia	`npm restart`
inizio	`npm start`
Stop	`npm start`
aggiornare	`npm update`
versione	`npm version`

Examples

Installazione dei pacchetti

introduzione

Pacchetto è un termine utilizzato da npm per indicare gli strumenti che gli sviluppatori possono utilizzare per i loro progetti. Questo include tutto, dalle librerie e framework come jQuery e AngularJS ai runner di attività come Gulp.js. I pacchetti arriveranno in una cartella chiamata tipicamente `node_modules` , che conterrà anche un file `package.json` . Questo file contiene informazioni su tutti i pacchetti comprese eventuali dipendenze, che sono moduli aggiuntivi necessari per utilizzare un particolare pacchetto.

Npm utilizza la riga di comando per installare e gestire i pacchetti, quindi gli utenti che tentano di utilizzare npm dovrebbero avere familiarità con i comandi di base sul loro sistema operativo, ovvero: attraversare le directory e poter vedere il contenuto delle directory.

Installazione di NPM

Si noti che per installare i pacchetti, è necessario aver installato NPM.

Il modo consigliato per installare NPM è quello di utilizzare uno dei programmi di installazione dalla pagina di download Node.js. Puoi verificare se hai già installato node.js eseguendo il `npm -v` o `npm version` .

Dopo aver installato NPM tramite l'installer Node.js, assicurati di verificare la disponibilità di aggiornamenti. Questo perché NPM viene aggiornato più frequentemente rispetto all'installer Node.js. Per verificare la presenza di aggiornamenti, eseguire il seguente comando:

```
npm install npm@latest -g
```

Come installare i pacchetti

Per installare uno o più pacchetti, utilizzare quanto segue:

```
npm install <package-name>
# or
npm i <package-name>...

# e.g. to install lodash and express
npm install lodash express
```

> **Nota** : questo installerà il pacchetto nella directory in cui si trova attualmente la riga di comando, quindi è importante verificare se è stata scelta la directory appropriata

Se hai già un file `package.json` nella directory di lavoro corrente e le dipendenze sono definite al suo interno, l' `npm install` risolverà e installerà automaticamente tutte le dipendenze elencate nel file. Puoi anche usare la versione abbreviata del comando di `npm install` che è: `npm i`

Se si desidera installare una versione specifica di un pacchetto, utilizzare:

```
npm install <name>@<version>

# e.g. to install version 4.11.1 of the package lodash
npm install lodash@4.11.1
```

Se si desidera installare una versione che corrisponde a un intervallo di versione specifico, utilizzare:

```
npm install <name>@<version range>

# e.g. to install a version which matches "version >= 4.10.1" and "version < 4.11.1"
# of the package lodash
npm install lodash@">=4.10.1 <4.11.1"
```

Se si desidera installare l'ultima versione, utilizzare:

```
npm install <name>@latest
```

I comandi sopra riportati cercheranno i pacchetti nel repository centrale di `npm` su npmjs.com . Se non stai cercando di installare dal registro di `npm` , sono supportate altre opzioni, come ad esempio:

```
# packages distributed as a tarball
npm install <tarball file>
npm install <tarball url>

# packages available locally
npm install <local path>

# packages available as a git repository
npm install <git remote url>

# packages available on GitHub
npm install <username>/<repository>

# packages available as gist (need a package.json)
npm install gist:<gist-id>

# packages from a specific repository
npm install --registry=http://myreg.mycompany.com <package name>

# packages from a related group of packages
# See npm scope
npm install @<scope>/<name>(@<version>)

# Scoping is useful for separating private packages hosted on private registry from
# public ones by setting registry for specific scope
npm config set @mycompany:registry http://myreg.mycompany.com
npm install @mycompany/<package name>
```

Di solito, i moduli verranno installati localmente in una cartella denominata `node_modules` , che può essere trovata nella directory di lavoro corrente. Questa è la directory `require()` che userà per caricare i moduli al fine di renderli disponibili.

Se hai già creato un file `package.json` , puoi utilizzare l' `--save` (abbreviazione `-S`) o una delle sue varianti per aggiungere automaticamente il pacchetto installato al tuo `package.json` come dipendenza. Se qualcun altro installa il pacchetto, `npm` leggerà automaticamente le dipendenze dal file `package.json` e installerà le versioni elencate. Nota che puoi ancora aggiungere e gestire le tue dipendenze modificando il file in un secondo momento, quindi di solito è una buona idea tenere traccia delle dipendenze, ad esempio usando:

```
npm install --save <name> # Install dependencies
# or
npm install -S <name> # shortcut version --save
# or
npm i -S <name>
```

Per installare i pacchetti e salvarli solo se sono necessari per lo sviluppo, non per eseguirli, non se sono necessari per l'esecuzione dell'applicazione, attenersi al seguente comando:

```
npm install --save-dev <name> # Install dependencies for development purposes
# or
npm install -D <name> # shortcut version --save-dev
# or
npm i -D <name>
```

Installare le dipendenze

Alcuni moduli non forniscono solo una libreria da utilizzare, ma forniscono anche uno o più binari che devono essere utilizzati tramite la riga di comando. Sebbene sia ancora possibile installare tali pacchetti localmente, è spesso preferibile installarli globalmente in modo da poter abilitare gli strumenti da riga di comando. In tal caso, `npm` collegherà automaticamente i file binari ai percorsi appropriati (ad esempio `/usr/local/bin/<name>`) in modo che possano essere utilizzati dalla riga di comando. Per installare un pacchetto a livello globale, utilizzare:

```
npm install --global <name>
# or
npm install -g <name>
# or
npm i -g <name>

# e.g. to install the grunt command line tool
npm install -g grunt-cli
```

Se si desidera visualizzare un elenco di tutti i pacchetti installati e le relative versioni associate nello spazio di lavoro corrente, utilizzare:

```
npm list
npm list <name>
```

L'aggiunta di un argomento di nome opzionale può controllare la versione di un pacchetto specifico.

Nota: se si verificano problemi di autorizzazione durante il tentativo di installare un modulo npm a livello globale, resistere alla tentazione di eseguire `sudo npm install -g ...` per risolvere il problema. La concessione di script di terze parti per l'esecuzione sul sistema con privilegi elevati è pericolosa. Il problema dell'autorizzazione potrebbe significare che hai un problema con il modo in `npm` è stato installato `npm` . Se sei interessato a installare il nodo in ambienti utente sandbox, potresti provare a utilizzare nvm .

Se si dispone di strumenti di compilazione o altre dipendenze di solo sviluppo (ad esempio Grunt), è possibile che non si desideri averli in bundle con l'applicazione distribuita. Se questo è il caso, ti consigliamo di avere una dipendenza di sviluppo, che è elencata in `package.json` sotto `devDependencies` . Per installare un pacchetto come dipendenza solo per lo sviluppo, usa `--save-dev` (o `-D`).

```
npm install --save-dev <name> // Install development dependencies which is not included in
```

```
production
# or
npm install -D <name>
```

Vedrai che il pacchetto viene quindi aggiunto alle `devDependencies` del tuo `package.json` .

Per installare le dipendenze di un progetto node.js scaricato / clonato, puoi semplicemente usarlo

```
npm install
# or
npm i
```

npm legge automaticamente le dipendenze da `package.json` e le installa.

NPM dietro un server proxy

Se l'accesso a Internet avviene tramite un server proxy, potrebbe essere necessario modificare i comandi di installazione di npm che accedono ai repository remoti. npm utilizza un file di configurazione che può essere aggiornato tramite la riga di comando:

```
npm config set
```

Puoi individuare le tue impostazioni proxy dal pannello delle impostazioni del browser. Dopo aver ottenuto le impostazioni del proxy (URL del server, porta, nome utente e password); è necessario configurare le configurazioni di npm come segue.

```
$ npm config set proxy http://<username>:<password>@<proxy-server-url>:<port>
$ npm config set https-proxy http://<username>:<password>@<proxy-server-url>:<port>
```

`username` , `password` , campi della `port` sono opzionali. Una volta impostati questi, l' `npm install` `npm i -g` , `npm i -g` ecc. `npm i -g` correttamente.

Ambiti e depositi

```
# Set the repository for the scope "myscope"
npm config set @myscope:registry http://registry.corporation.com

# Login at a repository and associate it with the scope "myscope"
npm adduser --registry=http://registry.corporation.com --scope=@myscope

# Install a package "mylib" from the scope "myscope"
npm install @myscope/mylib
```

Se il nome del proprio pacchetto inizia con `@myscope` e l'ambito "myscope" è associato a un repository differente, `npm publish` caricherà il pacchetto su quel repository.

Puoi anche mantenere queste impostazioni in un file `.npmrc` :

```
@myscope:registry=http://registry.corporation.com
//registry.corporation.com/:_authToken=xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxx
```

Questo è utile quando si automatizza la build su un server CI fe

Disinstallazione dei pacchetti

Per disinstallare uno o più pacchetti installati localmente, utilizzare:

```
npm uninstall <package name>
```

Il comando di disinstallazione per npm ha cinque alias che possono essere utilizzati anche:

```
npm remove <package name>
npm rm <package name>
npm r <package name>

npm unlink <package name>
npm un <package name>
```

Se si desidera rimuovere il pacchetto dal file `package.json` come parte della disinstallazione, utilizzare il flag `--save` (abbreviazione: `-S`):

```
npm uninstall --save <package name>
npm uninstall -S <package name>
```

Per una dipendenza di sviluppo, usa il `--save-dev` (abbreviazione: `-D`):

```
npm uninstall --save-dev <package name>
npm uninstall -D <package name>
```

Per una dipendenza opzionale, usa il `--save-optional` (abbreviazione: `-O`):

```
npm uninstall --save-optional <package name>
npm uninstall -O <package name>
```

Per i pacchetti installati a livello globale utilizzare il flag `--global` (abbreviazione: `-g`):

```
npm uninstall -g <package name>
```

Controllo delle versioni semantiche di base

Prima di pubblicare un pacchetto devi metterlo in versione. npm supporta il versioning semantico , questo significa che ci sono **patch,** versioni **minori e principali** .

Ad esempio, se il tuo pacchetto è alla versione 1.2.3 per cambiare versione devi:

1. patch release: `npm version patch` => 1.2.4
2. versione minore: `npm version minor` => 1.3.0

3. major release: `npm version major` => 2.0.0

Puoi anche specificare una versione direttamente con:

`npm version 3.1.4` => 3.1.4

Quando imposti una versione del pacchetto usando uno dei comandi npm sopra, npm modificherà il campo versione del file package.json, lo impegnerà e creerà anche un nuovo tag Git con la versione preceduta da una "v", come se tu Ho emesso il comando:

```
git tag v3.1.4
```

A differenza di altri gestori di pacchetti come Bower, il registro di npm non si basa su tag Git creati per ogni versione. Ma, se ti piace usare i tag, dovresti ricordare di inserire il tag appena creato dopo aver scaricato la versione del pacchetto:

`git push origin master` (per inviare la modifica a package.json)

`git push origin v3.1.4` (per spingere il nuovo tag)

Oppure puoi farlo in un colpo solo con:

```
git push origin master --tags
```

Impostazione di una configurazione del pacchetto

Le configurazioni del pacchetto Node.js sono contenute in un file chiamato `package.json` che puoi trovare alla radice di ogni progetto. È possibile impostare un nuovo file di configurazione chiamando:

```
npm init
```

Questo cercherà di leggere la directory di lavoro corrente per informazioni sul repository Git (se esiste) e le variabili di ambiente per provare e completare automaticamente alcuni dei valori segnaposto per te. In caso contrario, fornirà una finestra di dialogo di input per le opzioni di base.

Se desideri creare un `package.json` con i valori predefiniti, utilizza:

```
npm init --yes
# or
npm init -y
```

Se stai creando un `package.json` per un progetto che non stai pubblicando come pacchetto npm (vale a dire esclusivamente per arrotondare le tue dipendenze), puoi trasmettere questo intento nel tuo file `package.json` :

1. Facoltativamente, imposta la proprietà `private` su true per impedire la pubblicazione accidentale.
2. Opzionalmente imposta la proprietà della `license` su "NON LICENZIATO" per negare agli altri il diritto di utilizzare il tuo pacchetto.

Per installare un pacchetto e salvarlo automaticamente nel pacchetto `package.json` , utilizzare:

```
npm install --save <package>
```

Il pacchetto e i metadati associati (come la versione del pacchetto) appariranno nelle dipendenze. Se si salva se come dipendenza di sviluppo (usando `--save-dev`), il pacchetto apparirà invece nelle proprie `devDependencies` .

Con questo `package.json` bare-bones, si incontreranno messaggi di avvertimento durante l'installazione o l'aggiornamento di pacchetti, indicando che mancano una descrizione e il campo del repository. Mentre è sicuro ignorare questi messaggi, puoi eliminarli aprendo il pacchetto. Json in qualsiasi editor di testo e aggiungendo le seguenti linee all'oggetto JSON:

```
[...]
"description": "No description",
"repository": {
  "private": true
},
[...]
```

Pubblicare un pacchetto

Innanzitutto, assicurati di aver configurato il tuo pacchetto (come indicato in Configurazione di una configurazione del pacchetto). Quindi, devi accedere a npmjs.

Se hai già un utente npm

```
npm login
```

Se non hai un utente

```
npm adduser
```

Per verificare che l'utente sia registrato nel client corrente

```
npm config ls
```

Dopodiché, quando il pacchetto è pronto per essere pubblicato, usa

```
npm publish
```

E hai finito.

Se è necessario pubblicare una nuova versione, assicurarsi di aggiornare la versione del pacchetto, come indicato nella versione semantica di base . Altrimenti, `npm` non ti permetterà di pubblicare il pacchetto.

```
{
```

```
    name: "package-name",
    version: "1.0.4"
}
```

Esecuzione di script

È possibile definire script nel `package.json` , ad esempio:

```
{
  "name": "your-package",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "",
  "license": "ISC",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "echo": "echo hello!"
  }
}
```

Per eseguire lo script `echo` , eseguire `npm run echo` dalla riga di comando. Gli script arbitrari, come `echo` sopra, devono essere eseguiti con `npm run <script name>` . npm ha anche una serie di script ufficiali che vengono eseguiti in determinate fasi della vita del pacchetto (come `preinstall`). Vedi qui per l'intera panoramica di come npm gestisce i campi di script.

Gli script npm sono usati più spesso per cose come l'avvio di un server, la costruzione del progetto e l'esecuzione di test. Ecco un esempio più realistico:

```
  "scripts": {
    "test": "mocha tests",
    "start": "pm2 start index.js"
  }
```

Nelle voci degli `scripts` , i programmi da riga di comando come `mocha` funzioneranno se installati globalmente o localmente. Se la voce della riga di comando non esiste nel PATH di sistema, npm controllerà anche i pacchetti installati localmente.

Se i tuoi script diventano molto lunghi, possono essere suddivisi in parti, come questo:

```
  "scripts": {
    "very-complex-command": "npm run chain-1 && npm run chain-2",
    "chain-1": "webpack",
    "chain-2": "node app.js"
  }
```

Rimozione di pacchetti estranei

Per rimuovere pacchetti estranei (i pacchetti installati ma non nell'elenco delle dipendenze), eseguire il seguente comando:

```
npm prune
```

Per rimuovere tutti i pacchetti `dev` aggiungere `--production` flag di produzione:

```
npm prune --production
```

Altro su di esso

Elenco dei pacchetti attualmente installati

Per generare una lista (vista ad albero) dei pacchetti attualmente installati, usare

```
npm list
```

ls , **la** e **ll** sono alias di comando **lista** . i comandi la e ll mostrano informazioni estese come descrizione e repository.

Opzioni

Il formato della risposta può essere modificato passando le opzioni.

```
npm list --json
```

- **json** - Mostra le informazioni in formato json
- **lunga** - Mostra informazioni estese
- **parseable** - Mostra l'elenco parseable invece dell'albero
- **globale** : mostra i pacchetti installati globalmente
- **depth** - Massima profondità di visualizzazione dell'albero delle dipendenze
- **dev** / **development** - Mostra devDependencies
- **prod** / **production** - Mostra dipendenze

Se vuoi, puoi anche andare alla pagina principale del pacchetto.

```
npm home <package name>
```

Aggiornamento di npm e pacchetti

Poiché npm stesso è un modulo Node.js, può essere aggiornato utilizzando se stesso.

Se il sistema operativo è Windows, deve essere in esecuzione il prompt dei comandi come amministratore

```
npm install -g npm@latest
```

Se vuoi verificare le versioni aggiornate puoi fare:

```
npm outdated
```

Per aggiornare un pacchetto specifico:

```
npm update <package name>
```

> Questo aggiornerà il pacchetto alla versione più recente in base alle restrizioni in package.json

Nel caso in cui si desideri bloccare anche la versione aggiornata in package.json:

```
npm update <package name> --save
```

Bloccare i moduli su versioni specifiche

Per impostazione predefinita, npm installa l'ultima versione disponibile dei moduli in base alla versione semantica delle dipendenze. Questo può essere problematico se un autore di un modulo non aderisce a semere e introduce cambiamenti di rottura in un aggiornamento del modulo, per esempio.

Per bloccare la versione di ogni dipendenza (e le versioni delle relative dipendenze, ecc.) `node_modules` versione specifica installata localmente nella cartella `node_modules` , utilizzare

```
npm shrinkwrap
```

Questo creerà quindi un `npm-shrinkwrap.json` insieme al `package.json` che elenca le versioni specifiche delle dipendenze.

Impostazione per pacchetti installati globalmente

Puoi usare `npm install -g` per installare un pacchetto "globalmente". In genere ciò viene fatto per installare un eseguibile che è possibile aggiungere al percorso da eseguire. Per esempio:

```
npm install -g gulp-cli
```

Se aggiorni il tuo percorso, puoi chiamare direttamente `gulp` .

Su molti sistemi operativi, `npm install -g` tenterà di scrivere in una directory che l'utente potrebbe non essere in grado di scrivere come `/usr/bin` . **Non** si deve usare `sudo npm install` in questo caso poiché non v'è un possibile rischio per la sicurezza di eseguire script arbitrari con `sudo` e l'utente root può creare directory nella vostra casa che non è possibile scrivere al quale rende installazioni future più difficile.

Puoi dire a `npm` dove installare i moduli globali tramite il tuo file di configurazione, `~/.npmrc` . Questo è chiamato il `prefix` che è possibile visualizzare con il `npm prefix` .

```
prefix=~/.npm-global-modules
```

Questo utilizzerà il prefisso ogni volta che si esegue `npm install -g` . È anche possibile utilizzare

`npm install --prefix ~/.npm-global-modules` per impostare il prefisso durante l'installazione. Se il prefisso è uguale alla configurazione, non è necessario utilizzare `-g` .

Per poter utilizzare il modulo installato globalmente, deve essere sul tuo percorso:

```
export PATH=$PATH:~/.npm-global-modules/bin
```

Ora quando esegui `npm install -g gulp-cli` sarai in grado di usare `gulp` .

Nota: quando si `npm install` (senza `-g`) il prefisso sarà la directory con `package.json` o la directory corrente se non si trova nessuno nella gerarchia. Questo crea anche una directory `node_modules/.bin` che ha gli eseguibili. Se si desidera utilizzare un eseguibile specifico per un progetto, non è necessario utilizzare `npm install -g` . È possibile utilizzare quello in `node_modules/.bin` .

Collegamento di progetti per il debugging e lo sviluppo più rapidi

Le dipendenze dei progetti di costruzione possono a volte essere un compito noioso. Invece di pubblicare una versione del pacchetto su NPM e installare la dipendenza per testare le modifiche, usa il `npm link` . `npm link` crea un `npm link` simbolico in modo che l'ultimo codice possa essere testato in un ambiente locale. Ciò rende più semplice testare gli strumenti globali e le dipendenze dei progetti, consentendo l'esecuzione del codice più recente prima di creare una versione pubblicata.

Testo guida

```
NAME
       npm-link - Symlink a package folder

SYNOPSIS
         npm link (in package dir)
         npm link [<@scope>/]<pkg>[@<version>]

         alias: npm ln
```

Passi per il collegamento delle dipendenze del progetto

Quando si crea il collegamento delle dipendenze, si noti che il nome del pacchetto è ciò che verrà referenziato nel progetto principale.

1. CD in una directory delle dipendenze (es: `cd ../my-dep`)
2. `npm link`
3. CD nel progetto che utilizzerà la dipendenza
4. `npm link my-dep` o if namespace `npm link @namespace/my-dep`

Passi per il collegamento di uno strumento globale

1. CD nella directory del progetto (es: `cd eslint-watch`)
2. `npm link`
3. Usa lo strumento
4. `esw --quiet`

Problemi che possono sorgere

I progetti di collegamento a volte possono causare problemi se la dipendenza o lo strumento globale sono già installati. `npm uninstall (-g) <pkg>` e quindi il `npm link` normalmente risolve qualsiasi problema che possa sorgere.

Leggi npm online: https://riptutorial.com/it/node-js/topic/482/npm

Capitolo 84: nvm - Node Version Manager

Osservazioni

Gli URL utilizzati negli esempi precedenti fanno riferimento a una versione specifica di Node Version Manager. È molto probabile che l'ultima versione sia diversa da quella a cui viene fatto riferimento. Per installare nvm usando l'ultima versione, clicca qui per accedere a nvm su GitHub, che ti fornirà gli ultimi URL.

Examples

Installa NVM

Puoi usare `curl` :

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Oppure puoi usare `wget` :

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Controlla la versione NVM

Per verificare che nvm sia stato installato, fare:

```
command -v nvm
```

che dovrebbe produrre 'nvm' se l'installazione ha avuto successo.

Installazione di una versione specifica del nodo

Elenco delle versioni remote disponibili per l'installazione

```
nvm ls-remote
```

Installazione di una versione remota

```
nvm install <version>
```

Per esempio

```
nvm install 0.10.13
```

Utilizzando una versione del nodo già installata

Per elencare le versioni locali disponibili del nodo tramite NVM:

```
nvm ls
```

Ad esempio, se `nvm ls` restituisce:

```
$ nvm ls
     v4.3.0
     v5.5.0
```

È possibile passare alla `v5.5.0` con:

```
nvm use v5.5.0
```

Installa nvm su Mac OSX

PROCESSO DI INSTALLAZIONE

È possibile installare Node Version Manager utilizzando git, curl o wget. Esegui questi comandi in **Terminale** su **Mac OSX** .

esempio di arricciatura:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

esempio di wget:

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

PROVA CHE NVM È STATO INSTALLATO CORRETTAMENTE

Per verificare che nvm sia stato installato correttamente, chiudere e riaprire Terminal e immettere `nvm` . Se ottieni un messaggio **nvm: comando non trovato** , il tuo sistema operativo potrebbe non avere il file **.bash_profile** necessario. In Terminale, inserisci `touch ~/.bash_profile` ed esegui nuovamente lo script di installazione sopra.

Se ottieni ancora **nvm: comando non trovato** , prova quanto segue:

- In Terminale, inserisci `nano .bashrc` . Dovresti vedere uno script di esportazione quasi identico al seguente:

  export NVM_DIR = "/ Users / johndoe / .nvm" [-s "$ NVM_DIR / nvm.sh"] &&. “$ NVM_DIR / nvm.sh”

- Copia lo script di esportazione e rimuovilo da **.bashrc**
- Salva e chiudi il file .bashrc (CTRL + O - Invio - CTRL + X)
- Successivamente, inserisci `nano .bash_profile` per aprire il profilo Bash

- Incolla lo script di esportazione copiato nel Profilo Bash su una nuova riga
- Salva e chiudi il profilo Bash (CTRL + O - Invio - CTRL + X)
- Infine, inserisci `nano .bashrc` per riaprire il file **.bashrc**
- Incolla la seguente riga nel file:

  source ~ / .nvm / nvm.sh

- Salva e chiudi (CTRL + O - Invio - CTRL + X)
- Riavvia Terminal e inserisci `nvm` per testare se funziona

Impostazione dell'alias per la versione del nodo

Se si desidera impostare un nome alias nella versione del nodo installato, eseguire:

```
nvm alias <name> <version>
```

Similare a unalias, fare:

```
nvm unalias <name>
```

Se si desidera impostare un'altra versione rispetto alla versione stabile come alias predefinito, potrebbe essere un caso appropriato. `default` versioni alias `default` vengono caricate sulla console per impostazione predefinita.

Piace:

```
nvm alias default 5.0.1
```

Quindi ogni volta che inizia **console / terminale** 5.0.1 sarebbe presente per impostazione predefinita.

Nota:

```
nvm alias # lists all aliases created on nvm
```

Esegui qualsiasi comando arbitrario in una subshell con la versione desiderata del nodo

Elenca tutte le versioni del nodo installate

```
nvm ls
    v4.5.0
    v6.7.0
```

Esegui il comando utilizzando qualsiasi versione installata del nodo

```
nvm run 4.5.0 --version or nvm exec 4.5.0 node --version
Running node v4.5.0 (npm v2.15.9)
```

```
v4.5.0
```

```
nvm run 6.7.0 --version or nvm exec 6.7.0 node --version
Running node v6.7.0 (npm v3.10.3)
v6.7.0
```

---

usando alias

```
nvm run default --version or nvm exec default node --version
Running node v6.7.0 (npm v3.10.3)
v6.7.0
```

Per installare la versione LTS del nodo

```
nvm install --lts
```

Cambio di versione

```
nvm use v4.5.0 or nvm use stable ( alias )
```

Leggi nvm - Node Version Manager online: https://riptutorial.com/it/node-js/topic/2823/nvm---node-version-manager

Capitolo 85: OAuth 2.0

Examples

OAuth 2 con implementazione Redis - grant_type: password

In questo esempio userò oauth2 in rest api con il database redis

> **Importante:** è necessario installare il database redis sul proprio computer, scaricarlo da qui per gli utenti linux e da qui installare la versione di Windows, e utilizzeremo l'app desktop di redis manager, installarla da qui .

Ora dobbiamo impostare il nostro server node.js per utilizzare il database redis.

- **Creazione del file del server: app.js**

```
    var express = require('express'),
  bodyParser = require('body-parser'),
  oauthserver = require('oauth2-server'); // Would be: 'oauth2-server'

var app = express();

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

app.use(bodyParser.json());

app.oauth = oauthserver({
  model: require('./routes/Oauth2/model'),
  grants: ['password', 'refresh_token'],
  debug: true
});

// Handle token grant requests
app.all('/oauth/token', app.oauth.grant());

app.get('/secret', app.oauth.authorise(), function (req, res) {
  // Will require a valid access_token
  res.send('Secret area');
});

app.get('/public', function (req, res) {
  // Does not require an access_token
  res.send('Public area');
});

// Error handling
app.use(app.oauth.errorHandler());

app.listen(3000);
```

- **Crea il modello Oauth2 nelle rotte / Oauth2 / model.js**

```
 var model = module.exports,
  util = require('util'),
  redis = require('redis');

var db = redis.createClient();

var keys = {
  token: 'tokens:%s',
  client: 'clients:%s',
  refreshToken: 'refresh_tokens:%s',
  grantTypes: 'clients:%s:grant_types',
  user: 'users:%s'
};

model.getAccessToken = function (bearerToken, callback) {
  db.hgetall(util.format(keys.token, bearerToken), function (err, token) {
    if (err) return callback(err);

    if (!token) return callback();

    callback(null, {
      accessToken: token.accessToken,
      clientId: token.clientId,
      expires: token.expires ? new Date(token.expires) : null,
      userId: token.userId
    });
  });
};

model.getClient = function (clientId, clientSecret, callback) {
  db.hgetall(util.format(keys.client, clientId), function (err, client) {
    if (err) return callback(err);

    if (!client || client.clientSecret !== clientSecret) return callback();

    callback(null, {
      clientId: client.clientId,
      clientSecret: client.clientSecret
    });
  });
};

model.getRefreshToken = function (bearerToken, callback) {
  db.hgetall(util.format(keys.refreshToken, bearerToken), function (err, token) {
    if (err) return callback(err);

    if (!token) return callback();

    callback(null, {
      refreshToken: token.accessToken,
      clientId: token.clientId,
      expires: token.expires ? new Date(token.expires) : null,
      userId: token.userId
    });
  });
};

model.grantTypeAllowed = function (clientId, grantType, callback) {
  db.sismember(util.format(keys.grantTypes, clientId), grantType, callback);
};
```

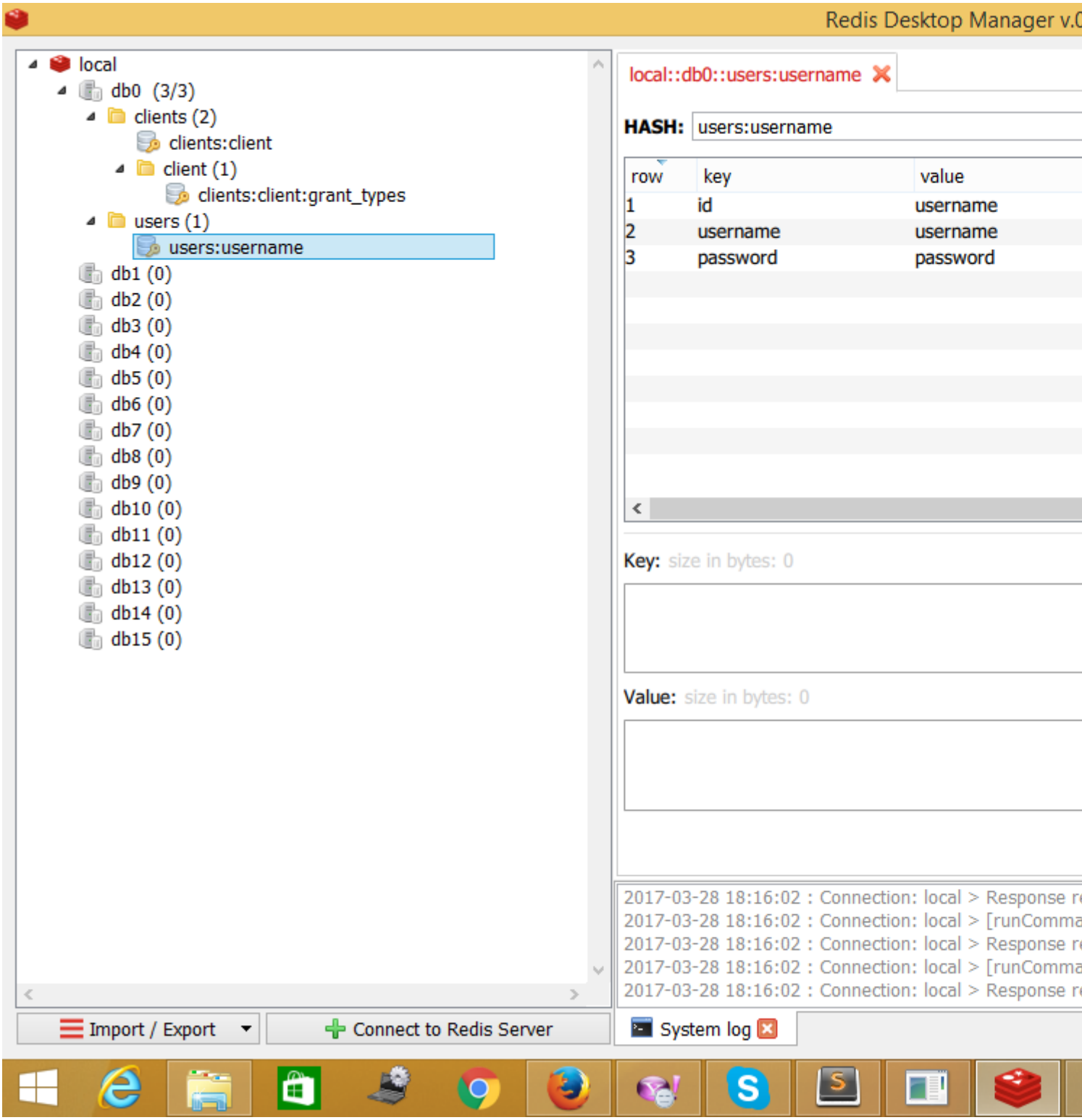
```
model.saveAccessToken = function (accessToken, clientId, expires, user, callback) {
  db.hmset(util.format(keys.token, accessToken), {
    accessToken: accessToken,
    clientId: clientId,
    expires: expires ? expires.toISOString() : null,
    userId: user.id
  }, callback);
};

model.saveRefreshToken = function (refreshToken, clientId, expires, user, callback) {
  db.hmset(util.format(keys.refreshToken, refreshToken), {
    refreshToken: refreshToken,
    clientId: clientId,
    expires: expires ? expires.toISOString() : null,
    userId: user.id
  }, callback);
};

model.getUser = function (username, password, callback) {
  db.hgetall(util.format(keys.user, username), function (err, user) {
    if (err) return callback(err);

    if (!user || password !== user.password) return callback();

    callback(null, {
      id: username
    });
  });
};
```

Devi solo installare redis sul tuo computer ed eseguire il seguente file del nodo

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

var db = require('redis').createClient();

db.multi()
  .hmset('users:username', {
    id: 'username',
    username: 'username',
    password: 'password'
  })
  .hmset('clients:client', {
    clientId: 'client',
    clientSecret: 'secret'
  })//clientId + clientSecret to base 64 will generate Y2xpZW50OnNlY3JldA==
  .sadd('clients:client:grant_types', [
    'password',
    'refresh_token'
  ])
  .exec(function (errs) {
    if (errs) {
      console.error(errs[0].message);
      return process.exit(1);
    }

    console.log('Client and user added successfully');
    process.exit();
  });
```

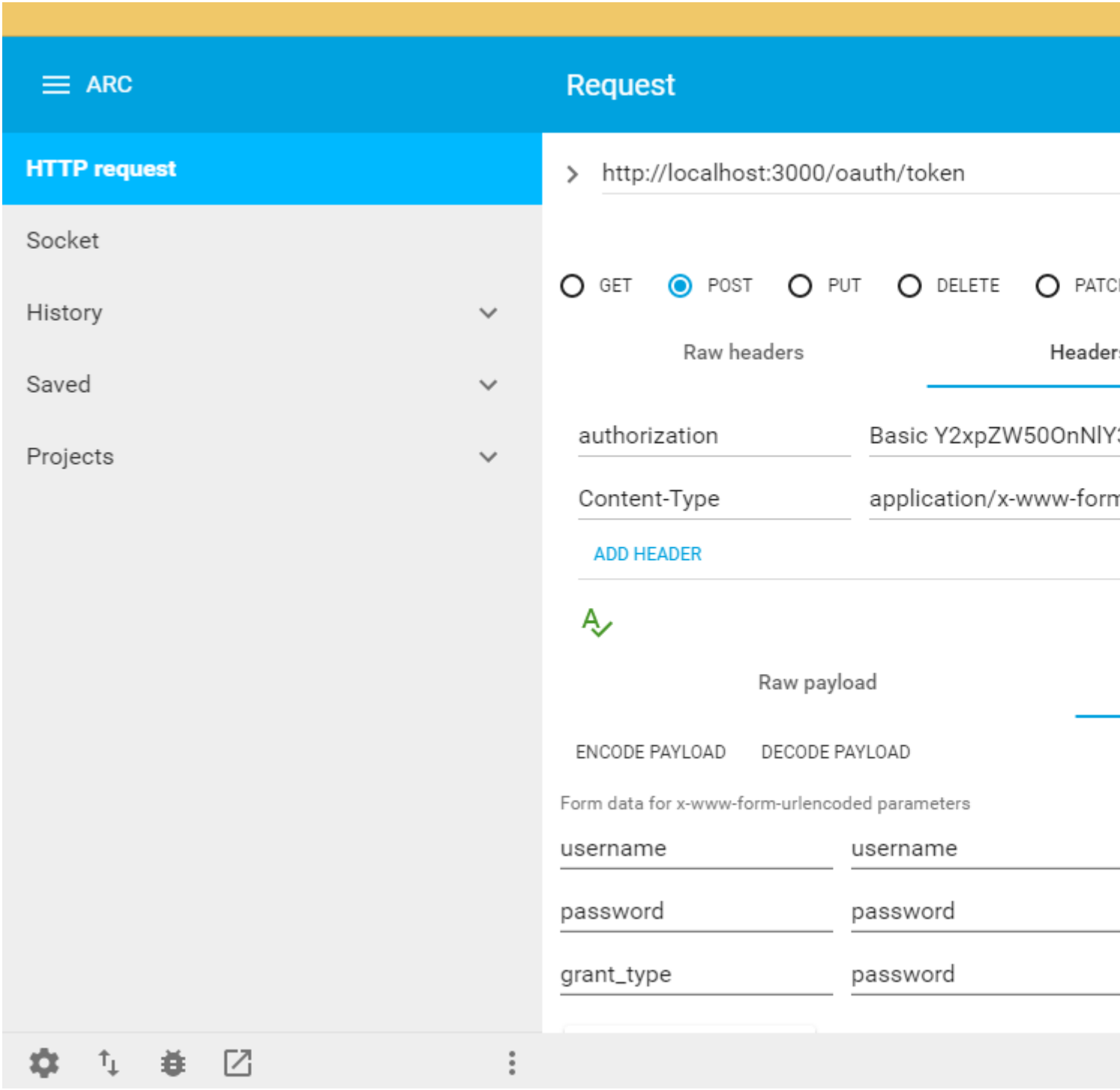
Nota : questo file imposterà le credenziali per il tuo frontend per richiedere il token. Quindi la tua richiesta da

Esempio di database redis dopo aver chiamato il file sopra:



La richiesta sarà la seguente:

Esempio di chiamata a API



Intestazione:

1. autorizzazione: base seguita dalla password impostata al primo avvio del ripristino:

   un. clientId + secretId a base64

2. Modulo dati

   username: utente che richiede token

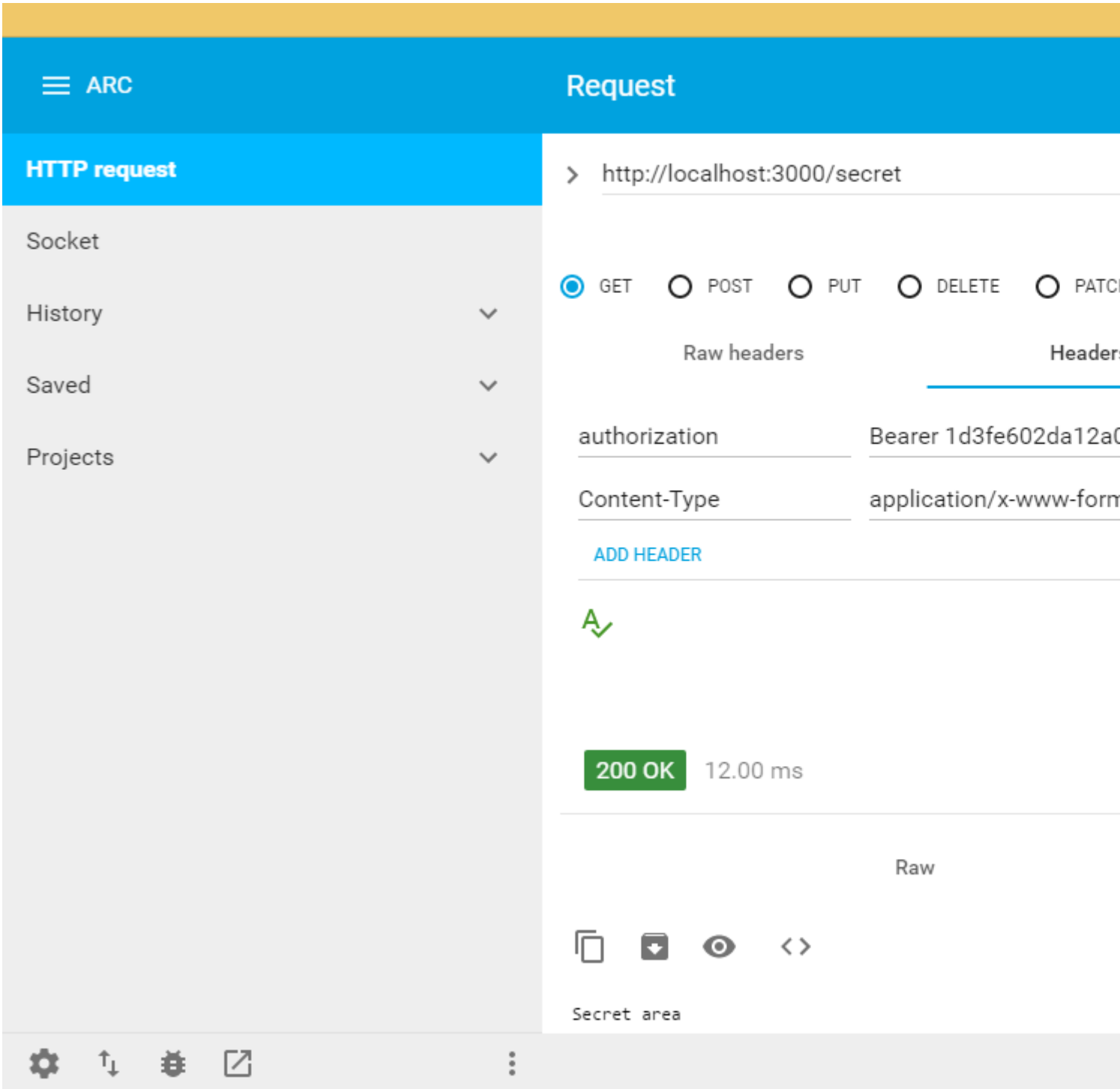
   password: password dell'utente

   grant_type: dipende da quali opzioni vuoi, scelgo passwod che richiede solo

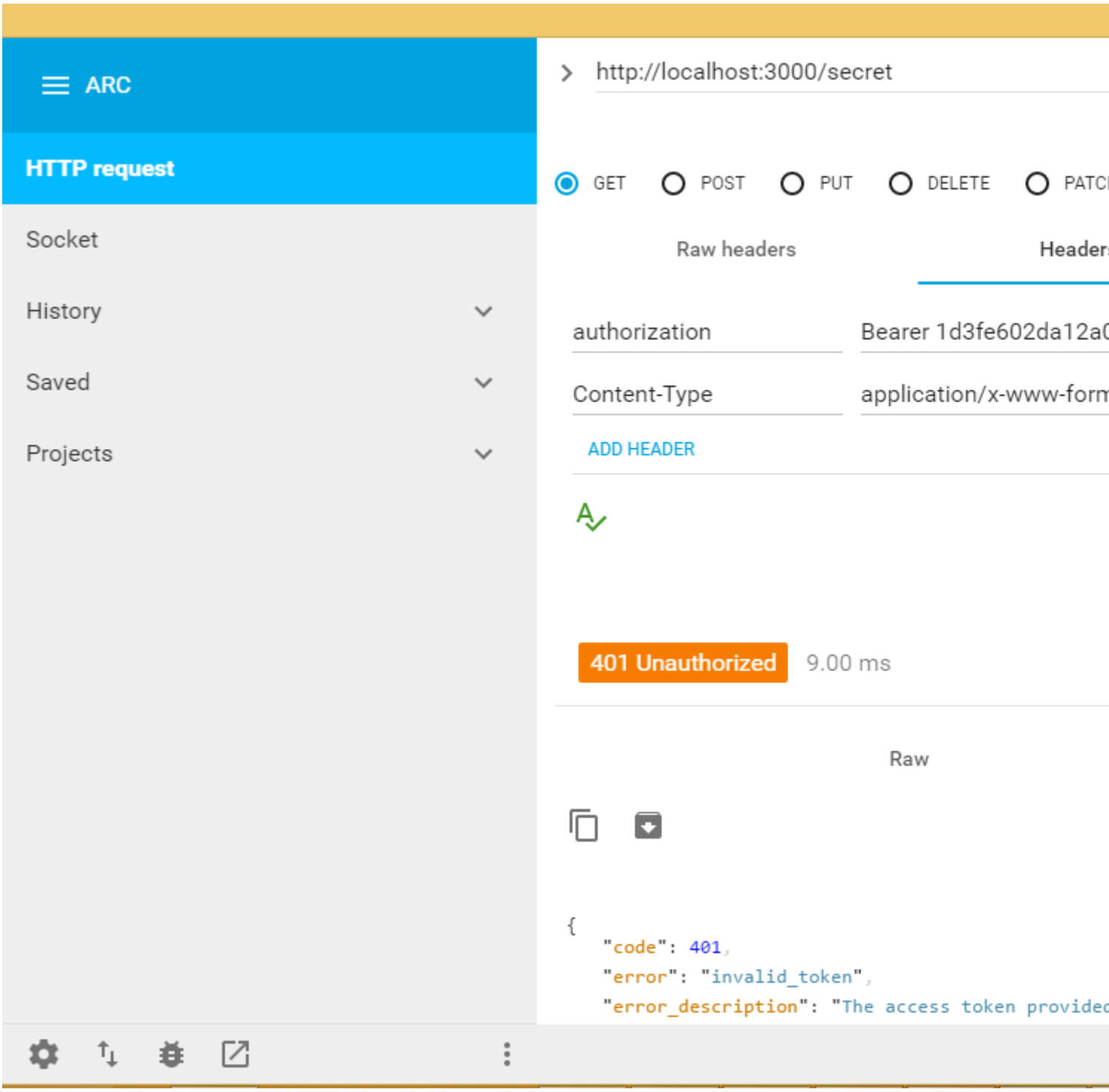
username e password per essere redistiti, i dati su redis saranno i seguenti:

```
{
  "access_token":"1d3fe602da12a086ecb2b996fd7b7ae874120c4f",
  "token_type":"bearer", // Will be used to access api + access+token e.g. bearer
1d3fe602da12a086ecb2b996fd7b7ae874120c4f
  "expires_in":3600,
  "refresh_token":"b6ad56e5c9aba63c85d7e21b1514680bbf711450"
}
```

Quindi dobbiamo chiamare la nostra API e prendere alcuni dati protetti con il nostro token di accesso che abbiamo appena creato, vedi sotto:

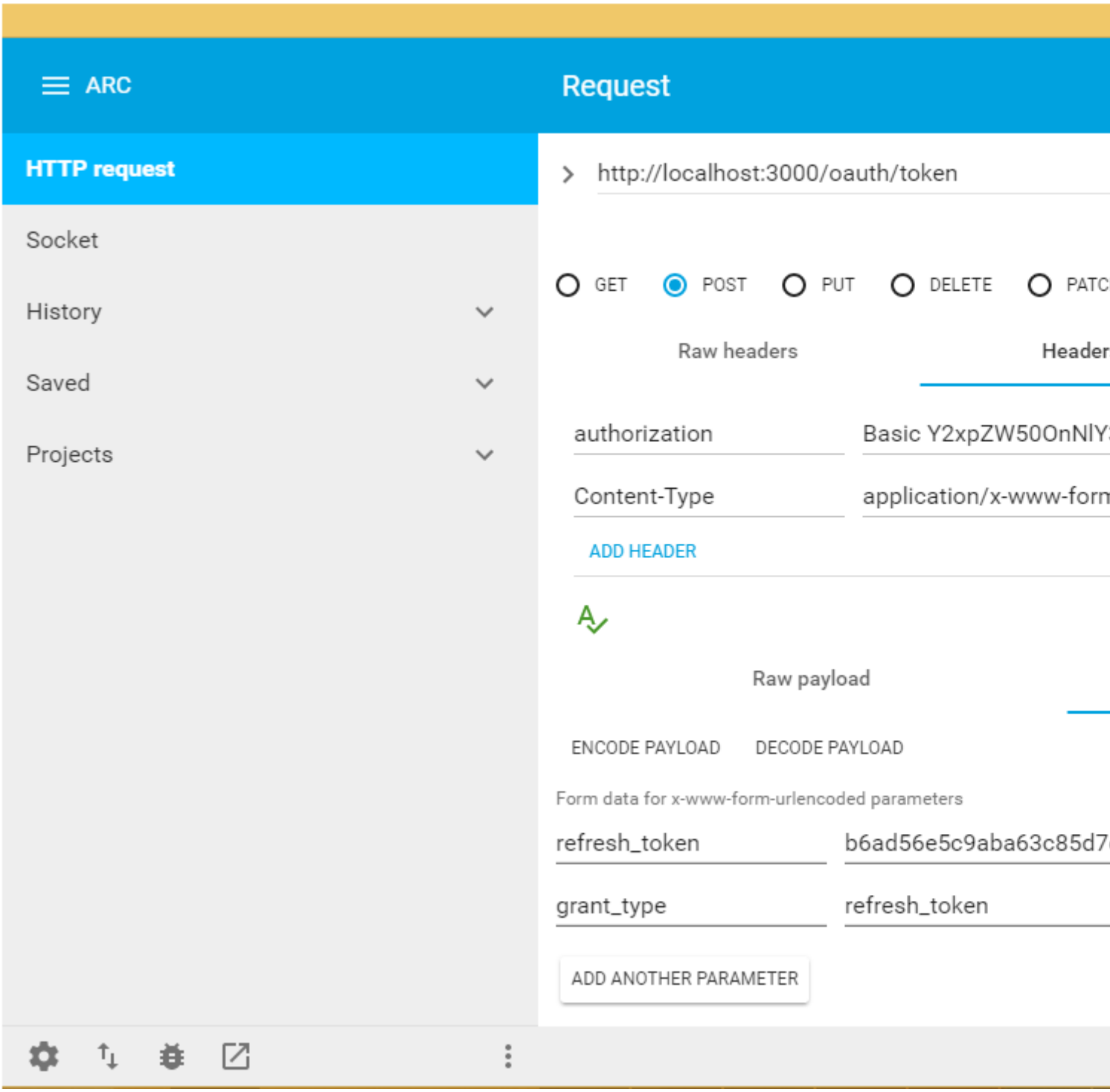


quando il token scade, api genera un errore che il token scade e non è possibile accedere a nessuna delle chiamate API, vedere l'immagine seguente:



> Vediamo cosa fare se il token scade, permettetemi prima di spiegarvelo, se il token di accesso scade un refresh_token esiste in redis che fa riferimento al token di accesso scaduto Quindi quello di cui abbiamo bisogno è chiamare oauth / token di nuovo con il grant_type refresh_token e impostare il autorizzazione al clientId di base: clientsecret (per basare 64!) e infine inviare il refresh_token, questo genererà un nuovo access_token con nuovi dati di scadenza.

L'immagine seguente mostra come ottenere un nuovo token di accesso:



Spero di aiutare!

Leggi OAuth 2.0 online: https://riptutorial.com/it/node-js/topic/9566/oauth-2-0

Capitolo 86: package.json

Osservazioni

Puoi creare `package.json` con

```
npm init
```

che ti chiederà informazioni di base sui tuoi progetti, incluso l' identificatore della licenza .

Examples

Definizione del progetto di base

```
{
    "name": "my-project",
    "version": "0.0.1",
    "description": "This is a project.",
    "author": "Someone <someone@example.com>",
    "contributors": [{
        "name": "Someone Else",
        "email": "else@example.com"
    }],
    "keywords": ["improves", "searching"]
}
```

Campo	Descrizione
nome	un campo **obbligatorio** per un pacchetto da installare. Deve essere minuscolo, parola singola senza spazi. (Sono consentiti trattini e trattini bassi)
versione	un campo **obbligatorio** per la versione del pacchetto che utilizza la versione semantica .
descrizione	una breve descrizione del progetto
autore	specifica l'autore del pacchetto
contributori	una serie di oggetti, uno per ogni contributore
parole chiave	una serie di stringhe, questo aiuterà le persone a trovare il tuo pacchetto

dipendenze

"dipendenze": {"nome-modulo": "0.1.0"}

- **esatto** : `0.1.0` installerà quella versione specifica del modulo.
- **la versione minore più recente** : `^0.1.0` installerà la versione minore più recente, ad esempio `0.2.0` , ma non installerà un modulo con una versione maggiore maggiore, ad esempio `1.0.0`
- **patch più recente** : `0.1.x` o `~0.1.0` installerà la versione di patch più recente disponibile, ad esempio `0.1.4` , ma non installerà un modulo con versione maggiore o minore maggiore, ad es. `0.2.0` o `1.0.0` .
- **carattere jolly** : `*` installerà l'ultima versione del modulo.
- **repository git** : il seguente installerà un tarball dal ramo master di un repository git. Possono anche essere forniti `#sha` , `#tag` o `#branch` :
  - **GitHub** : `user/project` o `user/project#v1.0.0`
  - **url** : `git://gitlab.com/user/project.git` o `git://gitlab.com/user/project.git#develop`
- **percorso locale** : `file:../lib/project`

Dopo averli aggiunti al pacchetto package.json, utilizzare il comando `npm install` nella directory del progetto nel terminale.

devDependencies

```
"devDependencies": {
    "module-name": "0.1.0"
}
```

Per le dipendenze richieste solo per lo sviluppo, come test di estensioni per i proxy di stile. Queste dev-dependencies non verranno installate quando si esegue "npm install" in modalità produzione.

Script

È possibile definire script che possono essere eseguiti o attivati prima o dopo un altro script.

```
{
  "scripts": {
    "pretest": "scripts/pretest.js",
    "test": "scripts/test.js",
    "posttest": "scripts/posttest.js"
  }
}
```

In questo caso, puoi eseguire lo script eseguendo uno di questi comandi:

```
$ npm run-script test
$ npm run test
$ npm test
$ npm t
```

Script predefiniti

Nome script	Descrizione
prepublish	Esegui prima che il pacchetto sia pubblicato.
pubblicare, postpublish	Esegui dopo che il pacchetto è stato pubblicato.
preinstallare	Esegui prima che il pacchetto sia installato.
installa, postinstall	Esegui dopo l'installazione del pacchetto.
preinstallare, disinstallare	Esegui prima che il pacchetto venga disinstallato.
postuninstall	Esegui dopo che il pacchetto è stato disinstallato.
preversione, versione	Esegui prima di eseguire il bump della versione del pacchetto.
postversion	Esegui dopo aver eseguito il bump della versione del pacchetto.
pretest, test, post test	Esegui dal comando di `npm test`
prestop, stop, poststop	Esegui con il comando `npm stop`
prestart, start, poststart	Esegui dal comando di `npm start`
prerestart, restart, postrestart	Esegui dal comando di `npm restart`

Script definiti dall'utente

Puoi anche definire i tuoi script nello stesso modo in cui lo fai con gli script predefiniti:

```
{
  "scripts": {
    "preci": "scripts/preci.js",
    "ci": "scripts/ci.js",
    "postci": "scripts/postci.js"
  }
}
```

In questo caso, puoi eseguire lo script eseguendo uno di questi comandi:

```
$ npm run-script ci
$ npm run ci
```

Gli script definiti dall'utente supportano anche gli script *pre* e *post* , come mostrato nell'esempio sopra.

Definizione del progetto estesa

Alcuni degli attributi aggiuntivi sono analizzati dal sito Web di npm come `repository` , `bugs` o `homepage` e mostrati nel riquadro informazioni per questi pacchetti

```
{
  "main": "server.js",
  "repository" :  {
    "type": "git",
    "url": "git+https://github.com/<accountname>/<repositoryname>.git"
  },
  "bugs": {
    "url": "https://github.com/<accountname>/<repositoryname>/issues"
  },
  "homepage": "https://github.com/<accountname>/<repositoryname>#readme",
  "files": [
    "server.js", // source files
    "README.md", // additional files
    "lib" // folder with all included files
  ]
}
```

Campo	Descrizione
principale	Script di entrata per questo pacchetto. Questo script viene restituito quando un utente richiede il pacchetto.
deposito	Ubicazione e tipo del repository pubblico
bug	Bugtracker per questo pacchetto (es. Github)
homepage	Homepage per questo pacchetto o il progetto generale
File	Elenco di file e cartelle che devono essere scaricati quando un utente esegue l' `npm install <packagename>`

Esplorando package.json

Un file `package.json` , solitamente presente nella root del progetto, contiene metadati relativi all'app o al modulo e all'elenco delle dipendenze da installare da npm durante l'esecuzione `npm install` .

Per inizializzare un `package.json` digitare `npm init` nel prompt dei comandi.

Per creare un `package.json` con i valori predefiniti utilizzare:

```
npm init --yes
# or
npm init -y
```

Per installare un pacchetto e salvarlo su `package.json` usa:

```
npm install {package name} --save
```

Puoi anche usare la notazione abbreviata:

```
npm i -S {package name}
```

Alias NPM `-S` per `--save` e `-D` per `--save-dev` per salvare rispettivamente nelle dipendenze di produzione o di sviluppo.

Il pacchetto apparirà nelle dipendenze; se usi `--save-dev` anziché `--save` , il pacchetto apparirà nelle tue devDependencies.

Proprietà importanti di `package.json` :

```
{
  "name": "module-name",
  "version": "10.3.1",
  "description": "An example module to illustrate the usage of a package.json",
  "author": "Your Name <your.name@example.org>",
  "contributors": [{
    "name": "Foo Bar",
    "email": "foo.bar@example.com"
  }],
  "bin": {
    "module-name": "./bin/module-name"
  },
  "scripts": {
    "test": "vows --spec --isolate",
    "start": "node index.js",
    "predeploy": "echo About to deploy",
    "postdeploy": "echo Deployed",
    "prepublish": "coffee --bare --compile --output lib/foo src/foo/*.coffee"
  },
  "main": "lib/foo.js",
  "repository": {
    "type": "git",
    "url": "https://github.com/username/repo"
  },
  "bugs": {
    "url": "https://github.com/username/issues"
  },
  "keywords": [
    "example"
  ],
  "dependencies": {
    "express": "4.2.x"
  },
  "devDependencies": {
    "assume": "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"
  },
  "peerDependencies": {
    "moment": ">2.0.0"
  },
  "preferGlobal": true,
  "private": true,
  "publishConfig": {
    "registry": "https://your-private-hosted-npm.registry.domain.com"
  },
  "subdomain": "foobar",
  "analyze": true,
  "license": "MIT",
  "files": [
    "lib/foo.js"
```

```
  ]
}
```

Informazioni su alcune proprietà importanti:

```
name
```

Il nome univoco del tuo pacchetto e dovrebbe essere in minuscolo. Questa proprietà è richiesta e il pacchetto non verrà installato senza di esso.

1. Il nome deve essere inferiore o uguale a 214 caratteri.
2. Il nome non può iniziare con un punto o un trattino basso.
3. I nuovi pacchetti non devono contenere lettere maiuscole nel nome.

```
version
```

La versione del pacchetto è specificata da Semantic Versioning (semere). Che presuppone che un numero di versione sia scritto come MAJOR.MINOR.PATCH e si incrementa il:

1. Versione MAJOR quando si apportano modifiche API incompatibili
2. Versione MINORE quando si aggiungono funzionalità in modo compatibile con le versioni precedenti
3. Versione PATCH quando si effettuano correzioni di errori compatibili con le versioni precedenti

```
description
```

La descrizione del progetto. Cerca di mantenerlo breve e conciso.

```
author
```

L'autore di questo pacchetto.

```
bin
```

Un oggetto che viene utilizzato per esporre script binari dal pacchetto. L'oggetto presuppone che la chiave sia il nome dello script binario e il valore di un percorso relativo allo script.

Questa proprietà viene utilizzata dai pacchetti che contengono una CLI (interfaccia a riga di comando).

```
script
```

Un oggetto che espone ulteriori comandi di npm. L'oggetto presuppone che la chiave sia il comando npm e che il valore sia il percorso dello script. Questi script possono essere eseguiti quando si esegue `npm run {command name}` o `npm run-script {command name}` .

I pacchetti che contengono un'interfaccia a riga di comando e sono installati localmente possono essere chiamati senza un percorso relativo. Quindi, invece di chiamare `./node-modules/.bin/mocha` puoi chiamare direttamente `mocha` .

```
main
```

Il principale punto di accesso al tuo pacchetto. Quando si chiama `require('{module name}')` nel nodo, questo sarà il file effettivo richiesto.

È altamente consigliato che richiedere il file principale non generi effetti collaterali. Ad esempio, richiedendo il file principale non dovrebbe avviare un server HTTP o connettersi a un database. Invece, dovresti creare qualcosa come `exports.init = function () {...}` nel tuo script principale.

```
keywords
```

Una serie di parole chiave che descrivono il tuo pacchetto. Questi aiuteranno le persone a trovare il tuo pacco.

```
devDependencies
```

Queste sono le dipendenze destinate esclusivamente allo sviluppo e al test del modulo. Le dipendenze verranno installate automaticamente a meno che non sia stata impostata la variabile di ambiente di `NODE_ENV=production` . Se questo è il caso puoi ancora questi pacchetti usando `npm install --dev`

```
peerDependencies
```

Se stai usando questo modulo, peerDependencies elenca i moduli che devi installare insieme a questo. Ad esempio, il `moment-timezone` deve essere installato insieme al `moment` perché è un plugin per il momento, anche se non `require("moment")` direttamente `require("moment")` .

```
preferGlobal
```

Una proprietà che indica che questa pagina preferisce essere installata globalmente usando `npm install -g {module-name}` . Questa proprietà viene utilizzata dai pacchetti che contengono una CLI (interfaccia a riga di comando).

In tutte le altre situazioni NON dovresti usare questa proprietà.

```
publishConfig
```

PublishConfig è un oggetto con valori di configurazione che verranno utilizzati per la pubblicazione dei moduli. I valori di configurazione impostati sostituiscono la configurazione predefinita di npm.

L'uso più comune di `publishConfig` è di pubblicare il tuo pacchetto su un registro privato di npm in modo da avere ancora i vantaggi di npm ma per i pacchetti privati. Questo viene fatto semplicemente impostando l'URL del tuo npm privato come valore per la chiave di registro.

```
files
```

Questa è una matrice di tutti i file da includere nel pacchetto pubblicato. È possibile utilizzare un percorso di file o un percorso di cartella. Verranno inclusi tutti i contenuti di un percorso di cartella. Ciò riduce la dimensione totale del pacchetto includendo solo i file corretti da distribuire. Questo campo funziona in congiunzione con un file di regole `.npmignore` .

fonte

Leggi package.json online: https://riptutorial.com/it/node-js/topic/1515/package-json

Capitolo 87: parser csv nel nodo js

introduzione

La lettura dei dati da un CSV può essere gestita in molti modi. Una soluzione è leggere il file `csv` in un array. Da lì puoi lavorare sull'array.

Examples

Usare FS per leggere in un CSV

fs è l' API del file system nel nodo. Possiamo usare il metodo readFile sulla nostra variabile fs, passargli un file `data.csv` , il formato e la funzione che legge e divide il `csv` per ulteriori elaborazioni.

Questo presuppone che tu abbia un file denominato `data.csv` nella stessa cartella.

```
'use strict'

const fs = require('fs');

fs.readFile('data.csv', 'utf8', function (err, data) {
  var dataArray = data.split(/\r?\n/);
  console.log(dataArray);
});
```

Ora puoi usare l'array come qualsiasi altro per lavorare su di esso.

Leggi parser csv nel nodo js online: https://riptutorial.com/it/node-js/topic/9162/parser-csv-nel-nodo-js

Capitolo 88: passport.js

introduzione

Passport è un popolare modulo di autorizzazione per il nodo. In poche parole gestisce tutte le richieste di autorizzazione sulla tua app da parte degli utenti. Passport supporta oltre 300 strategie in modo da poter facilmente integrare il login con Facebook / Google o qualsiasi altro social network che lo utilizza. La strategia di cui parleremo qui è il Local in cui si autentica un utente utilizzando il proprio database di utenti registrati (utilizzando nome utente e password).

Examples

Esempio di LocalStrategy in passport.js

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;

passport.serializeUser(function(user, done) { //In serialize user you decide what to store in
the session. Here I'm storing the user id only.
  done(null, user.id);
});

passport.deserializeUser(function(id, done) { //Here you retrieve all the info of the user
from the session storage using the user id stored in the session earlier using serialize user.
  db.findById(id, function(err, user) {
    done(err, user);
    });
});

passport.use(new LocalStrategy(function(username, password, done) {
    db.findOne({'username':username},function(err,student){
        if(err)return done(err,{message:message});//wrong roll_number or password;
        var pass_retrieved = student.pass_word;
        bcrypt.compare(password, pass_retrieved, function(err3, correct) {
          if(err3){
            message = [{"msg": "Incorrect Password!"}];
            return done(null,false,{message:message});  // wrong password
          }
          if(correct){
              return done(null,student);
          }
        });
    });
}));

app.use(session({ secret: 'super secret' })); //to make passport remember the user on other
pages too.(Read about session store. I used express-sessions.)
app.use(passport.initialize());
app.use(passport.session());

app.post('/',passport.authenticate('local',{successRedirect:'/users' failureRedirect: '/'}),
    function(req,res,next){
});
```

Leggi passport.js online: https://riptutorial.com/it/node-js/topic/8812/passport-js

Capitolo 89: Pool di connessione Mysql

Examples

Utilizzo di un pool di connessioni senza database

Raggiungimento della multi-tenancy sul server di database con più database ospitati su di esso.

La multitenancy è un requisito comune delle applicazioni aziendali al giorno d'oggi e la creazione di un pool di connessioni per ogni database nel server di database non è consigliata. quindi, ciò che possiamo fare è creare un pool di connessioni con un server di database e passare da un database all'altro su un server di database su richiesta.

Supponiamo che la nostra applicazione abbia diversi database per ogni azienda ospitata sul server di database. Ci collegheremo al rispettivo database aziendale quando l'utente raggiunge l'applicazione. ecco l'esempio su come farlo: -

```
var pool  = mysql.createPool({
      connectionLimit : 10,
      host            : 'example.org',
      user            : 'bobby',
      password        : 'pass'
    });

pool.getConnection(function(err, connection){
    if(err){
        return cb(err);
    }
    connection.changeUser({database : "firm1"});
    connection.query("SELECT * from history", function(err, data){
        connection.release();
        cb(err, data);
    });
});
```

Lasciatemi fare un esempio:

Durante la definizione della configurazione del pool non ho fornito il nome del database, ma ho fornito solo il server del database

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

quindi, quando vogliamo utilizzare il database specifico sul server di database, chiediamo la connessione al database dei risultati utilizzando: -

```
    connection.changeUser({database : "firm1"});
```

puoi consultare la documentazione ufficiale qui

Leggi Pool di connessione Mysql online: https://riptutorial.com/it/node-js/topic/6353/pool-di-connessione-mysql

Capitolo 90: Prestazioni Node.js

Examples

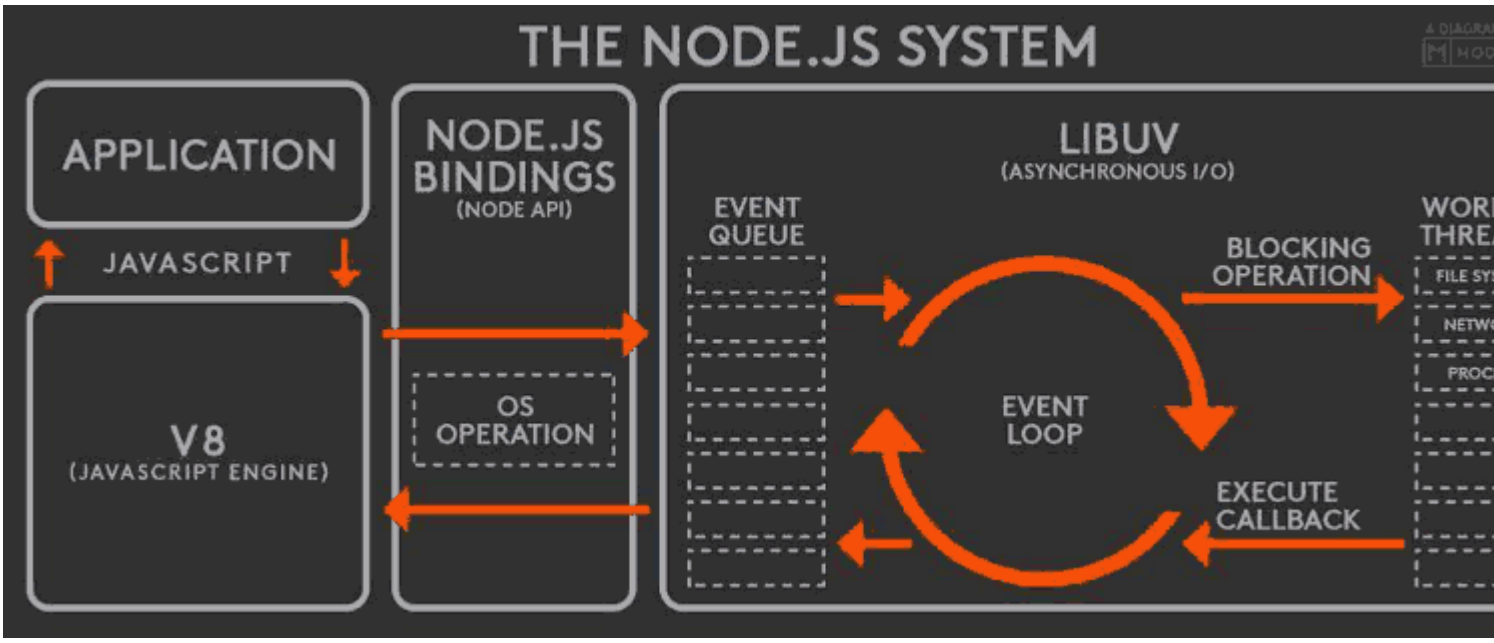
Loop degli eventi

Esempio di operazione di blocco

```
let loop = (i, max) => {
  while (i < max) i++
  return i
}

// This operation will block Node.js
// Because, it's CPU-bound
// You should be careful about this kind of code
loop(0, 1e+12)
```

Esempio di operazione IO non bloccante

```
let i = 0

const step = max => {
  while (i < max) i++
  console.log('i = %d', i)
}

const tick = max => process.nextTick(step, max)

// this will postpone tick run step's while-loop to event loop cycles
// any other IO-bound operation (like filesystem reading) can take place
// in parallel
tick(1e+6)
tick(1e+7)
console.log('this will output before all of tick operations. i = %d', i)
console.log('because tick operations will be postponed')
tick(1e+8)
```



In termini più semplici, Event Loop è un meccanismo di coda a thread singolo che esegue il codice associato alla CPU fino alla fine della sua esecuzione e al codice associato all'IO in modo non bloccante.

Tuttavia, Node.js sotto il tappeto utilizza multi-threading per alcune delle sue operazioni attraverso la libreria libuv .

Considerazioni sulle prestazioni

- Le operazioni non bloccanti non bloccheranno la coda e non influenzeranno le prestazioni del ciclo.
- Tuttavia, le operazioni associate alla CPU bloccheranno la coda, quindi dovresti fare attenzione a non eseguire operazioni legate alla CPU nel tuo codice Node.js.

Node.js non blocca l'IO perché scarica il lavoro nel kernel del sistema operativo e quando l'operazione IO fornisce i dati (*come evento*), notificherà il tuo codice con i callback forniti.

Aumenta max socket

Nozioni di base

```
require('http').globalAgent.maxSockets = 25

// You can change 25 to Infinity or to a different value by experimenting
```

Node.js utilizza per impostazione predefinita `maxSockets = Infinity` allo stesso tempo (dalla versione 1.0.0.0). Fino al nodo v0.12.0, il valore predefinito era `maxSockets = 5` (vedere v0.11.0). Quindi, dopo più di 5 richieste, verranno accodate. Se si desidera la concorrenza, aumentare questo numero.

Impostare il proprio agente

`http` API `http` utilizza un " **Agente globale** " . Puoi fornire il tuo agente. Come questo:

```
const http = require('http')
const myGloriousAgent = new http.Agent({ keepAlive: true })
myGloriousAgent.maxSockets = Infinity

http.request({ ..., agent: myGloriousAgent }, ...)
```

Disattivare completamente Socket Pooling

```
const http = require('http')
const options = {.....}

options.agent = false

const request = http.request(options)
```

insidie

- Dovresti fare la stessa cosa per l'API `https` se vuoi gli stessi effetti
- Attenzione, ad esempio, AWS utilizzerà 50 anziché `Infinity` .

Abilita gzip

```
const http = require('http')
const fs   = require('fs')
const zlib = require('zlib')

http.createServer((request, response) => {
  const stream          = fs.createReadStream('index.html')
  const acceptsEncoding = request.headers['accept-encoding']

  let encoder = {
    hasEncoder     : false,
    contentEncoding: {},
    createEncoder  : () => throw 'There is no encoder'
  }

  if (!acceptsEncoding) {
    acceptsEncoding = ''
  }

  if (acceptsEncoding.match(/\bdeflate\b/)) {
    encoder = {
      hasEncoder     : true,
      contentEncoding: { 'content-encoding': 'deflate' },
      createEncoder  : zlib.createDeflate
```

```
    }
  } else if (acceptsEncoding.match(/\bgzip\b/)) {
    encoder = {
      hasEncoder     : true,
      contentEncoding: { 'content-encoding': 'gzip' },
      createEncoder  : zlib.createGzip
    }
  }

  response.writeHead(200, encoder.contentEncoding)

  if (encoder.hasEncoder) {
    stream = stream.pipe(encoder.createEncoder())
  }

  stream.pipe(response)

}).listen(1337)
```

Leggi Prestazioni Node.js online: https://riptutorial.com/it/node-js/topic/9410/prestazioni-node-js

Capitolo 91: Programmazione asincrona

introduzione

Il nodo è un linguaggio di programmazione in cui tutto può essere eseguito in modo asincrono. Di seguito è possibile trovare alcuni esempi e le cose tipiche del funzionamento asincrono.

Sintassi

- doSomething ([args], function ([argsCB]) {/ * fa qualcosa quando è fatto * /});
- doSomething ([args], ([argsCB]) => {/ * fa qualcosa quando viene fatto * /});

Examples

Funzioni di callback

Funzioni di callback in JavaScript

Le funzioni di callback sono comuni in JavaScript. Le funzioni di callback sono possibili in JavaScript perché le funzioni sono cittadini di prima classe .

Callback sincroni.

Le funzioni di callback possono essere sincrone o asincrone. Poiché le funzioni di callback asincrone possono essere più complesse, ecco un semplice esempio di una funzione di callback sincrono.

```
// a function that uses a callback named `cb` as a parameter
function getSyncMessage(cb) {
    cb("Hello World!");
}

console.log("Before getSyncMessage call");
// calling a function and sending in a callback function as an argument.
getSyncMessage(function(message) {
    console.log(message);
});
console.log("After getSyncMessage call");
```

L'output per il codice sopra riportato è:

```
> Before getSyncMessage call
> Hello World!
> After getSyncMessage call
```

Innanzitutto analizzeremo come viene eseguito il codice precedente. Questo è più per coloro che non comprendono già il concetto di callback se già lo capisci, sentiti libero di saltare questo paragrafo. Prima viene analizzato il codice e quindi la prima cosa interessante che si verifica è che viene eseguita la riga 6 che emette `Before getSyncMessage call` alla console. Quindi viene eseguita la riga 8 che chiama la funzione `getSyncMessage` invia in una funzione anonima come argomento per il parametro `cb` nella funzione `getSyncMessage` . L'esecuzione è ora eseguita all'interno della funzione `getSyncMessage` sulla riga 3 che esegue la funzione `cb` che è stata appena `getSyncMessage` , questa chiamata invia una stringa di argomento "Hello World" per il parametro denominato `message` nella funzione anonima passata. L'esecuzione quindi va alla riga 9 che registra `Hello World!` alla console. Quindi l'esecuzione passa attraverso il processo di uscita dal callstack (vedere anche) che colpisce la linea 10 quindi la linea 4 e infine torna alla riga 11.

Alcune informazioni da sapere sui callback in generale:

- La funzione che invii a una funzione come richiamata può essere chiamata zero volte, una volta o più volte. Tutto dipende dall'implementazione.
- La funzione di callback può essere chiamata in modo sincrono o asincrono e possibilmente in modo sincrono e asincrono.
- Proprio come le normali funzioni, i nomi che dai parametri alla tua funzione non sono importanti ma l'ordine è. Così, per esempio, sulla riga 8 il `message` parametri potrebbe essere stato chiamato `statement` , `msg` , o se non hai senso qualcosa di simile a `jellybean` . Quindi dovresti sapere quali parametri sono stati inviati alla tua callback in modo da poterli ottenere nell'ordine giusto con nomi propri.

Callback asincroni.

Una cosa da notare su JavaScript è che è sincrono di default, ma ci sono delle API date nell'ambiente (browser, Node.js, ecc.) Che potrebbero renderlo asincrono (c'è di più a riguardo qui).

Alcune cose comuni che sono asincrone negli ambienti JavaScript che accettano i callback:

- eventi
- setTimeout
- setInterval
- l'API di recupero
- promesse

Anche qualsiasi funzione che utilizza una delle funzioni sopra può essere avvolta con una funzione che richiede una richiamata e il callback sarebbe quindi una richiamata asincrona (sebbene il wrapping di una promessa con una funzione che richiede una richiamata sia probabilmente considerato un anti-pattern come ci sono più modi preferiti per gestire le promesse).

Quindi data questa informazione possiamo costruire una funzione asincrona simile alla precedente sincrona.

```
// a function that uses a callback named `cb` as a parameter
function getAsyncMessage(cb) {
    setTimeout(function () { cb("Hello World!") }, 1000);
}

console.log("Before getSyncMessage call");
// calling a function and sending in a callback function as an argument.
getAsyncMessage(function(message) {
    console.log(message);
});
console.log("After getSyncMessage call");
```

Che stampa quanto segue alla console:

```
> Before getSyncMessage call
> After getSyncMessage call
// pauses for 1000 ms with no output
> Hello World!
```

L'esecuzione della linea passa ai registri della linea 6 "Prima della chiamata getSyncMessage". Quindi l'esecuzione passa alla riga 8 chiamando getAsyncMessage con un callback per il parametro `cb` . Viene quindi eseguita la riga 3 che chiama setTimeout con un callback come primo argomento e il numero 300 come secondo argomento. `setTimeout` fa tutto ciò che fa e trattiene quel callback in modo che possa chiamarlo più tardi in 1000 millisecondi, ma dopo aver impostato il timeout e prima che metta in pausa i 1000 millisecondi restituisce l'esecuzione al punto in cui era stata interrotta, quindi va alla riga 4 , quindi la riga 11, quindi si interrompe per 1 secondo e setTimeout chiama la sua funzione di richiamata che riprende l'esecuzione alla riga 3 dove `getAsyncMessages` richiamato il callback `getAsyncMessages` con il valore "Hello World" per il `message` parametro che viene quindi registrato sulla console alla riga 9 .

Funzioni di callback in Node.js

NodeJS ha callback asincroni e in genere fornisce due parametri alle tue funzioni a volte convenzionalmente chiamati `err` e `data` . Un esempio con la lettura di un file di testo.

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

fs.readFile("./test.txt", "utf8", function(err, data) {
    if(err) {
        // handle the error
    } else {
        // process the file text given with data
    }
});
```

Questo è un esempio di callback che viene chiamato una volta sola.

È buona pratica gestire l'errore in qualche modo anche se lo si registra o lo si gira. Il resto non è necessario se si lancia o si torna e può essere rimosso per ridurre il rientro finché si interrompe l'esecuzione della funzione corrente nel se facendo qualcosa come il lancio o il ritorno.

Sebbene possa essere comune vedere `err` , i `data` potrebbero non essere sempre il caso in cui i callback useranno tale modello, è meglio guardare la documentazione.

Un altro esempio di callback proviene dalla libreria express (espressa 4.x):

```
// this code snippet was on http://expressjs.com/en/4x/api.html
const express = require('express');
const app = express();

// this app.get method takes a url route to watch for and a callback
// to call whenever that route is requested by a user.
app.get('/', function(req, res){
  res.send('hello world');
});

app.listen(3000);
```

Questo esempio mostra una richiamata che viene chiamata più volte. Il callback è fornito con due oggetti come parametri qui denominati come `req` e `res` questi nomi corrispondono rispettivamente a richiesta e risposta, e forniscono modi per visualizzare la richiesta in entrata e impostare la risposta che verrà inviata all'utente.

Come puoi vedere ci sono vari modi in cui un callback può essere usato per eseguire il codice sincrono e asincrono in JavaScript e le callback sono molto diffuse in tutto il codice JavaScript.

Esempio di codice

Domanda: Qual è l'output del codice qui sotto e perché?

```
setTimeout(function() {
    console.log("A");
}, 1000);

setTimeout(function() {
    console.log("B");
}, 0);

getDataFromDatabase(function(err, data) {
    console.log("C");
    setTimeout(function() {
        console.log("D");
    }, 1000);
});

console.log("E");
```

Uscita: questo è noto con certezza: `EBAD . C` è sconosciuto quando verrà registrato.

Spiegazione: Il compilatore non si fermerà sui `getDataFromDatabase setTimeout` e `getDataFromDatabase` . Quindi la prima riga che registrerà è `E` Le funzioni di callback *(primo argomento di `setTimeout`)* verranno eseguite dopo il timeout impostato su un modo asincrono!

Più dettagli:

1. `E` non ha `setTimeout`
2. `B` ha un timeout impostato di 0 millisecondi
3. `A` ha un timeout impostato di 1000 millisecondi
4. `D` deve richiedere un database, dopo che `D` deve attendere 1000 millisecondi quindi viene dopo `A`
5. `C` è sconosciuto perché è sconosciuto quando vengono richiesti i dati del database. Potrebbe essere prima o dopo `A`

Gestione degli errori asincroni

Prova a prendere

Gli errori devono sempre essere gestiti. Se si utilizza la programmazione sincrona, è possibile utilizzare un `try catch` . Ma questo non funziona se lavori asincroni! Esempio:

```
try {
    setTimeout(function() {
        throw new Error("I'm an uncaught error and will stop the server!");
    }, 100);
}
catch (ex) {
    console.error("This error will not be work in an asynchronous situation: " + ex);
}
```

Gli errori asincroni verranno gestiti solo all'interno della funzione di callback!

Possibilità di lavoro

v0.8

Gestori di eventi

Le prime versioni di Node.JS hanno un gestore di eventi.

```
process.on("UncaughtException", function(err, data) {
    if (err) {
        // error handling
    }
});
```

v0.8

domini

All'interno di un dominio, gli errori vengono rilasciati tramite gli emettitori di eventi. Usando questo sono tutti gli errori, i timer, i metodi di callback implicitamente registrati solo all'interno del dominio.

In caso di errore, inviare un evento di errore e non arrestare l'applicazione.

```
var domain = require("domain");
var d1 = domain.create();
var d2 = domain.create();

d1.run(function() {
    d2.add(setTimeout(function() {
        throw new Error("error on the timer of domain 2");
    }, 0));
});

d1.on("error", function(err) {
    console.log("error at domain 1: " + err);
});

d2.on("error", function(err) {
    console.log("error at domain 2: " + err);
});
```

Callback hell

L'inferno del callback (anche una piramide di doom o effetto boomerang) si verifica quando si annidano troppe funzioni di callback all'interno di una funzione di callback. Ecco un esempio per leggere un file (in ES6).

```
const fs = require('fs');
let filename = `${__dirname}/myfile.txt`;

fs.exists(filename, exists => {
    if (exists) {
        fs.stat(filename, (err, stats) => {
            if (err) {
                throw err;
            }
            if (stats.isFile()) {
                fs.readFile(filename, null, (err, data) => {
                    if (err) {
                        throw err;
                    }
                    console.log(data);
                });
            }
            else {
                throw new Error("This location contains not a file");
            }
        });
    }
    else {
        throw new Error("404: file not found");
    }
});
```

Come evitare "Callback Hell"

Si consiglia di annidare non più di 2 funzioni di callback. Questo ti aiuterà a mantenere la leggibilità del codice e sarà molto più facile mantenerlo in futuro. Se hai bisogno di nidificare più di

2 callback, prova a utilizzare gli eventi distribuiti .

Esiste anche una libreria chiamata async che aiuta a gestire i callback e la loro esecuzione disponibile su npm. Aumenta la leggibilità del codice di callback e offre un maggiore controllo sul flusso del codice di callback, incluso il permesso di eseguirli in parallelo o in serie.

Promesse native

v6.0.0

Le promesse sono uno strumento per la programmazione asincrona. In JavaScript promesse sono noti per le loro `then` metodi. Le promesse prevedono due stati principali "in attesa" e "risolti". Una volta che la promessa è 'risolta', non può tornare 'in sospeso'. Ciò significa che le promesse sono valide soprattutto per eventi che si verificano solo una volta. Lo stato 'sistemato' ha due stati pure 'risolti' e 'rifiutati'. Puoi creare una nuova promessa usando la `new` parola chiave e passando una funzione nel costruttore `new Promise(function (resolve, reject) {})` .

La funzione passata nel costruttore Promise riceve sempre un primo e un secondo parametro, solitamente denominati rispettivamente `resolve` e `reject` . La denominazione di questi due parametri è convenzione, ma metteranno la promessa nello stato 'risolto' o nello stato 'rifiutato'. Quando uno di questi è chiamato la promessa passa dall'essere "in sospeso" a "risolto". `resolve` viene chiamata quando l'azione desiderata, che è spesso asincrona, è stata eseguita e il `reject` viene utilizzato se l'azione è stata errata.

Nel sotto timeout è una funzione che restituisce una Promessa.

```
function timeout (ms) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve("It was resolved!");
    }, ms)
  });
}

timeout(1000).then(function (dataFromPromise) {
  // logs "It was resolved!"
  console.log(dataFromPromise);
})

console.log("waiting...");
```

uscita della console

```
waiting...
// << pauses for one second>>
It was resolved!
```

Quando viene chiamato timeout, la funzione passata al costruttore Promise viene eseguita senza ritardo. Quindi viene eseguito il metodo setTimeout e la richiamata viene attivata nei successivi `ms` millisecondi, in questo caso `ms=1000` . Poiché la richiamata al setTimeout non viene attivata, la funzione di timeout restituisce il controllo all'ambito della chiamata. La catena dei metodi `then`

viene quindi archiviata per essere chiamata più tardi quando / se la Promessa è stata risolta. Se esistessero metodi di `catch` , anche questi sarebbero archiviati, ma verrebbero licenziati quando / se la promessa 'respinge'.

Lo script stampa quindi "in attesa ...". Un secondo dopo, setTimeout chiama il suo callback che chiama la funzione di risoluzione con la stringa "È stato risolto!". Tale stringa viene quindi passata al callback del metodo `then` e quindi viene registrata all'utente.

Nello stesso senso è possibile racchiudere la funzione setTimeout asincrona che richiede una richiamata, quindi è possibile racchiudere qualsiasi azione asincrona singolare con una promessa.

Maggiori informazioni sulle promesse nella documentazione di JavaScript Promises .

Leggi Programmazione asincrona online: https://riptutorial.com/it/node-js/topic/8813/programmazione-asincrona

Capitolo 92: Programmazione sincrona contro asincrona in nodejs

Examples

Uso asincrono

Il pacchetto async fornisce funzioni per il codice asincrono.

Utilizzando la funzione automatica è possibile definire relazioni asincrone tra due o più funzioni:

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

async.auto({
    get_data: function(callback) {
        console.log('in get_data');
        // async code to get some data
        callback(null, 'data', 'converted to array');
    },
    make_folder: function(callback) {
        console.log('in make_folder');
        // async code to create a directory to store a file in
        // this is run at the same time as getting the data
        callback(null, 'folder');
    },
    write_file: ['get_data', 'make_folder', function(results, callback) {
        console.log('in write_file', JSON.stringify(results));
        // once there is some data and the directory exists,
        // write the data to a file in the directory
        callback(null, 'filename');
    }],
    email_link: ['write_file', function(results, callback) {
        console.log('in email_link', JSON.stringify(results));
        // once the file is written let's email a link to it...
        // results.write_file contains the filename returned by write_file.
        callback(null, {'file':results.write_file, 'email':'user@example.com'});
    }]
}, function(err, results) {
    console.log('err = ', err);
    console.log('results = ', results);
});
```

Questo codice potrebbe essere stato creato in modo sincrono, semplicemente chiamando `get_data` , `make_folder` , `write_file` e `email_link` nell'ordine corretto. Async tiene traccia dei risultati per te e, se si verifica un errore (primo parametro di `callback` non uguale a `null`), interrompe l'esecuzione delle altre funzioni.

Leggi Programmazione sincrona contro asincrona in nodejs online: https://riptutorial.com/it/node-js/topic/8287/programmazione-sincrona-contro-asincrona-in-nodejs

Capitolo 93: Promesse Bluebird

Examples

Conversione della libreria del nodoback in Promises

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

Promise.promisifyAll(fs)

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

Promesse funzionali

Esempio di mappa:

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

Esempio di filtro:

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

Esempio di riduzione:

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

Coroutine (generatori)

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

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

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

Smaltimento automatico delle risorse (Promise.using)

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

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

Eseguendo in serie

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

Leggi Promesse Bluebird online: https://riptutorial.com/it/node-js/topic/6728/promesse-bluebird

Capitolo 94: Protezione delle applicazioni Node.js

Examples

Prevenire la falsificazione di richieste cross-site (CSRF)

CSRF è un attacco che costringe l'utente finale a eseguire azioni indesiderate su un'applicazione web in cui è attualmente autenticato.

Può accadere perché i cookie vengono inviati con ogni richiesta a un sito Web, anche quando tali richieste provengono da un altro sito.

Possiamo usare il modulo `csurf` per creare il token csrf e convalidarlo.

Esempio

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

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

// create express app
var app = express()

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

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

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

Quindi, quando accediamo a `GET /form` , passeremo il token `csrfToken` alla vista.

Ora, all'interno della vista, imposta il valore csrfToken come valore di un campo di input nascosto chiamato `_csrf` .

ad esempio per `handlebar` modelli `handlebar`

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

```
    <button type="submit">Submit</button>
</form>
```

ad esempio per i modelli di `jade`

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

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

    input(type="submit")
```

ad esempio per i modelli di `ejs`

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

SSL / TLS in Node.js

Se si sceglie di gestire SSL / TLS nell'applicazione Node.js, considerare che si è anche responsabili della gestione della prevenzione degli attacchi SSL / TLS a questo punto. In molte architetture server-client, SSL / TLS termina su un proxy inverso, sia per ridurre la complessità delle applicazioni sia per ridurre l'ambito della configurazione della sicurezza.

Se l'applicazione Node.js deve gestire SSL / TLS, può essere protetta caricando i file key e cert.

Se il proprio fornitore di certificati richiede una catena di autorità di certificazione (CA), può essere aggiunto all'opzione `ca` come array. Una catena con più voci in un singolo file deve essere suddivisa in più file e inserita nello stesso ordine nell'array poiché Node.js attualmente non supporta più voci ca in un unico file. Un esempio è fornito nel seguente codice per i file `1_ca.crt` e `2_ca.crt` . Se l'array `ca` è richiesto e non impostato correttamente, i browser client potrebbero visualizzare messaggi che non potrebbero verificare l'autenticità del certificato.

Esempio

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

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

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

Utilizzando HTTPS

L'installazione minima per un server HTTPS in Node.js sarebbe qualcosa del genere:

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

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

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

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

Se vuoi anche supportare le richieste http, devi fare solo questa piccola modifica:

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

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

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

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

Configurare un server HTTPS

Una volta che node.js è installato sul sistema, basta seguire la procedura seguente per ottenere un server Web di base in esecuzione con supporto per HTTP e HTTPS!

Passaggio 1: creare un'autorità di certificazione

1. creare la cartella in cui si desidera memorizzare la chiave e il certificato:

   ```
   mkdir conf
   ```

---

2. vai a quella directory:

   ```
   cd conf
   ```

---

3. prendi questo file `ca.cnf` da usare come scorciatoia di configurazione:

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

---

4. creare una nuova autorità di certificazione utilizzando questa configurazione:

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

---

5. ora che abbiamo la nostra autorità di certificazione in `ca-key.pem` e `ca-cert.pem` , `ca-key.pem` una chiave privata per il server:

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

---

6. prendi questo file `server.cnf` da usare come scorciatoia di configurazione:

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

---

7. generare la richiesta di firma del certificato utilizzando questa configurazione:

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

---

8. firmare la richiesta:

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

Passaggio 2: installa il certificato come certificato di origine

1. copia il tuo certificato nella cartella dei certificati di root:

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

---

2. aggiornare l'archivio di CA:

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

Applicazione Secure express.js 3

La configurazione per creare una connessione sicura usando express.js (Dalla versione 3):

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

// Define your key and cert
```

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

// your express configuration here

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

// Using port 8080 for http and 8443 for https

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

In questo modo fornisci il middleware espresso al server http / https nativo

Se si desidera che l'app venga eseguita su porte inferiori a 1024, sarà necessario utilizzare il comando sudo (non consigliato) o utilizzare un proxy inverso (ad esempio nginx, haproxy).

Leggi Protezione delle applicazioni Node.js online: https://riptutorial.com/it/node-js/topic/3473/protezione-delle-applicazioni-node-js

Capitolo 95: Quadri di modelli

Examples

Nunjucks

Motore lato server con ereditarietà di blocchi, autoescaping, macro, controllo asincrono e altro. Molto ispirato da jinja2, molto simile a Twig (php).

Documenti - http://mozilla.github.io/nunjucks/
Installa - `npm i nunjucks`

Utilizzo di base con *Express di* seguito.

app.js

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

var app = express();
app.use(express.static('/public'));

// Apply nunjucks and add custom filter and function (for example).
var env = nunjucks.configure(['views/'], { // set folders with templates
    autoescape: true,
    express: app
});
env.addFilter('myFilter', function(obj, arg1, arg2) {
    console.log('myFilter', obj, arg1, arg2);
    // Do smth with obj
    return obj;
});
env.addGlobal('myFunc', function(obj, arg1) {
    console.log('myFunc', obj, arg1);
    // Do smth with obj
    return obj;
});

app.get('/', function(req, res){
    res.render('index.html', {title: 'Main page'});
});

app.get('/foo', function(req, res){
    res.locals.smthVar = 'This is Sparta!';
    res.render('foo.html', {title: 'Foo page'});
});

app.listen(3000, function() {
    console.log('Example app listening on port 3000...');
});
```

/views/index.html

```
<html>
<head>
    <title>Nunjucks example</title>
</head>
<body>
{% block content %}
{{title}}
{% endblock %}
</body>
</html>
```

/views/foo.html

```
{% extends "index.html" %}

{# This is comment #}
{% block content %}
    <h1>{{title}}</h1>
    {# apply custom function and next build-in and custom filters #}
    {{ myFunc(smthVar) | lower | myFilter(5, 'abc') }}
{% endblock %}
```

Leggi Quadri di modelli online: https://riptutorial.com/it/node-js/topic/5885/quadri-di-modelli

Capitolo 96: Quadri di test unitari

Examples

Mocha sincrono

```
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function() {
      expect([ 1, 2, 3 ].length).to.be.equal(3)
    })
  })
})
```

Mocha asincrono (callback)

```
var expect = require("chai").expect;
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function(done) {
      testSomething(err => {
        expect(err).to.not.be.equal(null)
        done()
      })
    })
  })
})
```

Mocha asincrono (Promessa)

```
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function() {
      return doSomething().then(result => {
         expect(result).to.be.equal('hello world')
      })
    })
  })
})
```

Mocha asincrono (async / await)

```
const { expect } = require('chai')

describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', async function() {
      const result = await answerToTheUltimateQuestion()
      expect(result).to.be.equal(42)
    })
  })
```

```
})
```

Leggi Quadri di test unitari online: https://riptutorial.com/it/node-js/topic/6731/quadri-di-test-unitari

Capitolo 97: Richiamata per promettere

Examples

Promuovere una richiamata

Richiamata a base di:

```
db.notification.email.find({subject: 'promisify callback'}, (error, result) => {
   if (error) {
       console.log(error);
   }

   // normal code here
});
```

Questo utilizza il promisifyAll metodo bluebird per promettere ciò che è convenzionalmente codice basato su callback come sopra. bluebird creerà una versione promettente di tutti i metodi nell'oggetto, i nomi dei metodi basati su promesse hanno accodati ad Async:

```
let email = bluebird.promisifyAll(db.notification.email);

email.findAsync({subject: 'promisify callback'}).then(result => {

    // normal code here
})
.catch(console.error);
```

Se solo i metodi specifici devono essere promessi, basta usare la sua promessa:

```
let find = bluebird.promisify(db.notification.email.find);

find({locationId: 168}).then(result => {

    // normal code here
});
.catch(console.error);
```

Ci sono alcune librerie (ad es., MassiveJS) che non possono essere promesse se l'oggetto immediato del metodo non viene passato sul secondo parametro. In tal caso, basta passare l'oggetto immediato del metodo che deve essere promesso sul secondo parametro e racchiuderlo nella proprietà context.

```
let find = bluebird.promisify(db.notification.email.find, { context: db.notification.email });

find({locationId: 168}).then(result => {

    // normal code here
});
.catch(console.error);
```

Promessa manuale di un callback

A volte potrebbe essere necessario promettere manualmente una funzione di callback. Potrebbe trattarsi di un caso in cui la richiamata non segue il formato standard di errore prima o se è necessaria una logica aggiuntiva per promettere:

Esempio con fs.exists (percorso, callback) :

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

var existsAsync = function(path) {
  return new Promise(function(resolve, reject) {
    fs.exists(path, function(exists) {
      // exists is a boolean
      if (exists) {
        // Resolve successfully
        resolve();
      } else {
        // Reject with error
        reject(new Error('path does not exist'));
      }
    });
});

// Use as a promise now
existsAsync('/path/to/some/file').then(function() {
  console.log('file exists!');
}).catch(function(err) {
  // file does not exist
  console.error(err);
});
```

setTimeout promesso

```
function wait(ms) {
    return new Promise(function (resolve, reject) {
        setTimeout(resolve, ms)
    })
}
```

Leggi Richiamata per promettere online: https://riptutorial.com/it/node-js/topic/2346/richiamata-per-promettere

Capitolo 98: Richiedere()

introduzione

Questa documentazione si concentra sulla spiegazione degli usi e della dichiarazione `require()` che NodeJS include nella loro lingua.

È necessario importare determinati file o pacchetti utilizzati con i moduli di NodeJS. È usato per migliorare la struttura del codice e gli usi. `require()` viene utilizzato su file installati localmente, con un percorso diretto dal file che è `require` .

Sintassi

- module.exports = {testFunction: testFunction};
- var test_file = require ('./ testFile.js'); // Lasciateci avere un file chiamato `testFile`
- test_file.testFunction (our_data); // Lascia che `testFile` abbia function `testFunction`

Osservazioni

L'uso di `require()` consente di strutturare il codice in modo simile all'utilizzo di classi e metodi pubblici da parte di Java. Se una funzione è `.export` ', può essere `require` ' ed in un altro file da utilizzare. Se un file non è `.export` ', non può essere utilizzato in un altro file.

Examples

Beginning require () usa con una funzione e un file

Require è una dichiarazione che Node interpreta come, in un certo senso, una funzione `getter` . Ad esempio, supponiamo di avere un file denominato `analysis.js` e l'aspetto interno del file è simile a questo,

```
function analyzeWeather(weather_data) {
  console.log('Weather information for ' + weather_data.time + ': ');
  console.log('Rainfall: ' + weather_data.precip);
  console.log('Temperature: ' + weather_data.temp);
  //More weather_data analysis/printing...
}
```

Questo file contiene solo il metodo, `analyzeWeather(weather_data)` . Se vogliamo usare questa funzione, deve essere usata all'interno di questo file, o copiata nel file che vuole essere usata da. Tuttavia, il nodo ha incluso uno strumento molto utile per aiutare con il codice e l'organizzazione dei file, che è moduli .

Per utilizzare la nostra funzione, dobbiamo prima `export` la funzione attraverso una dichiarazione all'inizio. Il nostro nuovo file ha questo aspetto,

```
module.exports = {
  analyzeWeather: analyzeWeather
}
function analyzeWeather(weather_data) {
  console.log('Weather information for ' + weather_data.time + ': ');
  console.log('Rainfall: ' + weather_data.precip);
  console.log('Temperature: ' + weather_data.temp);
  //More weather_data analysis/printing...
}
```

Con questa piccola istruzione `module.exports` , la nostra funzione è ora pronta per l'uso al di fuori del file. Tutto ciò che resta da fare è usare `require()` .

Quando si `require` una funzione o un file, la sintassi è molto simile. Di solito è fatto all'inizio del file e impostato su `var` o su `const` per l'uso in tutto il file. Ad esempio, abbiamo un altro file (sullo stesso livello di `analyze.js` chiamato `handleWeather.js` che assomiglia a questo,

```
const analysis = require('./analysis.js');

weather_data = {
  time: '01/01/2001',
  precip: 0.75,
  temp: 78,
  //More weather data...
};
analysis.analyzeWeather(weather_data);
```

In questo file, stiamo usando `require()` per prendere il nostro file `analysis.js` . Quando viene utilizzato, chiamiamo semplicemente la variabile o la costante assegnata a questa `require` e usiamo qualsiasi funzione all'interno di quella che viene esportata.

Beginning require () usa con un pacchetto NPM

Nodo di `require` è anche molto utile se usato in tandem con un pacchetto di NPM . Diciamo, per esempio, si desidera utilizzare il pacchetto NPM `require` in un file chiamato `getWeather.js` . Dopo l' installazione di NPM del pacchetto tramite la riga di comando (`git install request`), sei pronto per usarlo. Il `getWeather.js` file `getWeather.js` potrebbe sembrare come questo,

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

//Construct your url variable...
https.get(url, function(error, response, body) {
  if (error) {
    console.log(error);
  } else {
    console.log('Response => ' + response);
    console.log('Body => ' + body);
  }
});
```

Quando viene eseguito questo file, prima `require` (s) il pacchetto che hai appena installato chiamato `request` . All'interno del file di `request` , ci sono molte funzioni a cui puoi ora accedere, una delle quali è chiamata `get` . Nelle linee successive, la funzione viene utilizzata per fare una

richiesta HTTP GET .

Leggi Richiedere() online: https://riptutorial.com/it/node-js/topic/10742/richiedere--

Capitolo 99: Route-Controller-Struttura del servizio per ExpressJS

Examples

Modello-Routes-Controllers-Services Struttura delle directory

```
├──models
|   ├──user.model.js
├──routes
|   ├──user.route.js
├──services
|   ├──user.service.js
├──controllers
|   ├──user.controller.js
```

Per la struttura del codice modulare, la logica deve essere suddivisa in queste directory e file.

Modelli : la definizione dello schema del modello

Percorsi : l'API instrada le mappe ai Controller

Controllori : i controller gestiscono tutta la logica alla base della convalida dei parametri della richiesta, della query, dell'invio delle risposte con i codici corretti.

Servizi : i servizi contengono le query del database e restituiscono oggetti o errori di lancio

Questo codificatore finirà per scrivere più codici. Ma alla fine i codici saranno molto più manutenibili e separati.

Modello-Routes-Controllers-Services Struttura dei codici

user.model.js

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

const UserSchema  = new mongoose.Schema({
    name: String
})

const User = mongoose.model('User', UserSchema)

module.exports = User;
```

user.routes.js

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

var UserController = require('../controllers/user.controller')

router.get('/', UserController.getUsers)

module.exports = router;
```

user.controllers.js

```
var UserService = require('../services/user.service')

exports.getUsers = async function (req, res, next) {
    // Validate request parameters, queries using express-validator

    var page = req.params.page ? req.params.page : 1;
    var limit = req.params.limit ? req.params.limit : 10;
    try {
        var users = await UserService.getUsers({}, page, limit)
        return res.status(200).json({ status: 200, data: users, message: "Succesfully Users
Retrieved" });
    } catch (e) {
        return res.status(400).json({ status: 400, message: e.message });
    }
}
```

user.services.js

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

exports.getUsers = async function (query, page, limit) {

    try {
        var users = await User.find(query)
        return users;
    } catch (e) {
        // Log Errors
        throw Error('Error while Paginating Users')
    }
}
```

Leggi Route-Controller-Struttura del servizio per ExpressJS online: https://riptutorial.com/it/node-js/topic/10785/route-controller-struttura-del-servizio-per-expressjs

Capitolo 100: Routing NodeJs

introduzione

Come configurare il server Web di base Express sotto il nodo js e il router Exploring the Express.

Osservazioni

Infine, utilizzando Express Router è possibile utilizzare la funzione di routing nell'applicazione ed è facile da implementare.

Examples

Esecuzione del routing del server Web

Creazione di server Web Express

Il server Express è diventato pratico e utilizza molti utenti e community. Sta diventando popolare.

Consente di creare un server Express. Per la gestione dei pacchetti e la flessibilità per la dipendenza Useremo NPM (Node Package Manager).

1. Vai alla directory del progetto e crea il file package.json. ***package.json*** {"name": "expressRouter", "version": "0.0.1", "scripts": {"start": "node Server.js"}, "dependencies": {"express": "^ 4.12.3 "}}
2. Salvare il file e installare la dipendenza rapida utilizzando il seguente comando *npm install* . Questo creerà node_modules nella directory del progetto insieme alla dipendenza richiesta.
3. Creiamo Express Web Server. Vai alla directory del progetto e crea il file server.js. ***server.js***

   var express = require ("express"); var app = express ();

// Creazione dell'oggetto Router ()

var router = express.Router ();

// Fornisci tutti i percorsi qui, questo è per la Home page.

```
router.get("/",function(req,res){
res.json({"message" : "Hello World"});
```

});

app.use ("/ api", router);

// Ascolta questo porto

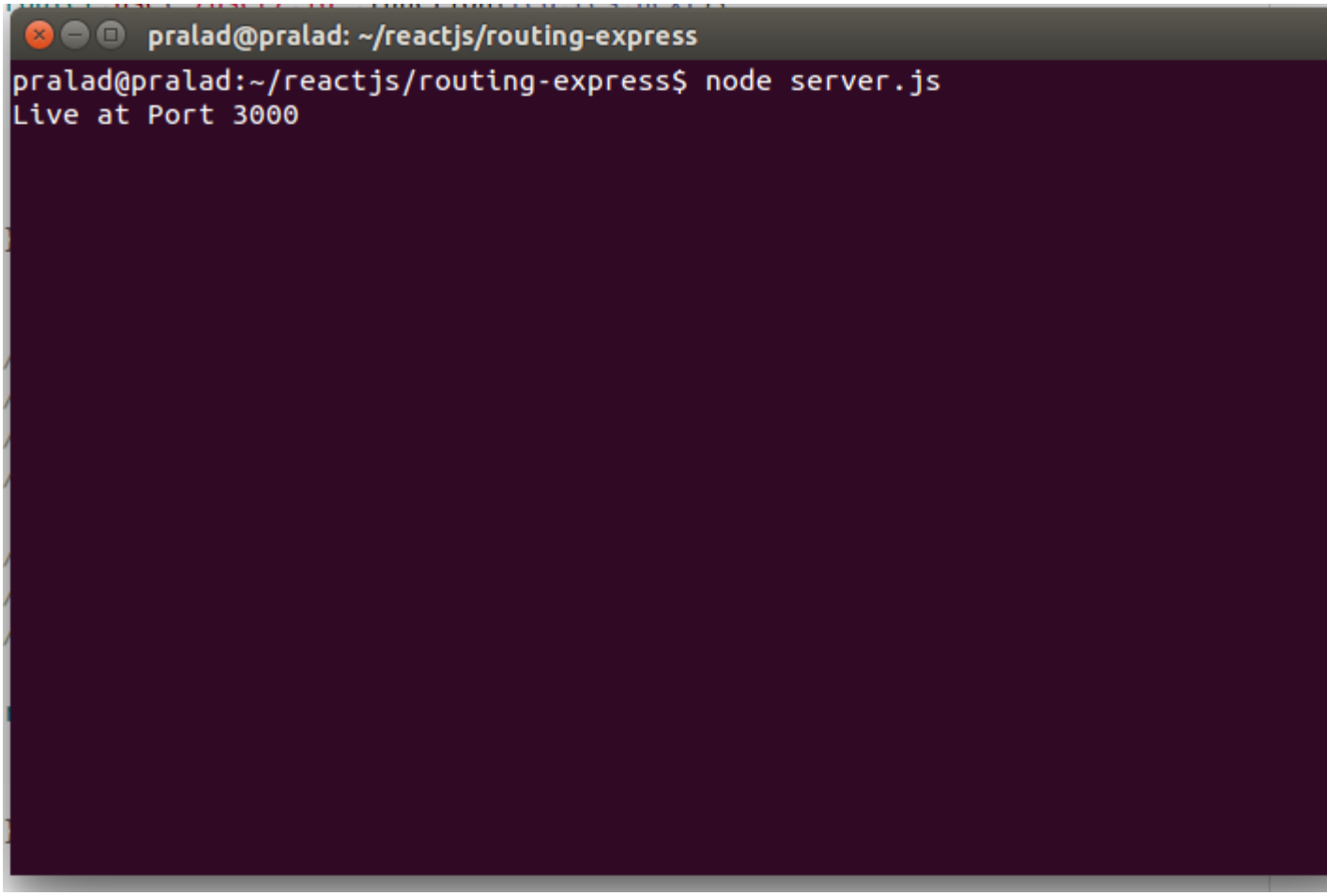
app.listen (3000, function () {console.log ("Live at Port 3000");});

```
For more detail on setting node server you can see [here][1].
```

4. Eseguire il server digitando il comando seguente.

   nodo server.js

   Se il server funziona correttamente, vedrai qualcosa di simile.



   .

5. Ora vai al browser o al postino e fai una richiesta

   http: // localhost: 3000 / api /

   L'output sarà

.

Questo è tutto, la base del routing espresso.

Ora gestiamo GET, POST, ecc.

Cambia il tuo file server.js come

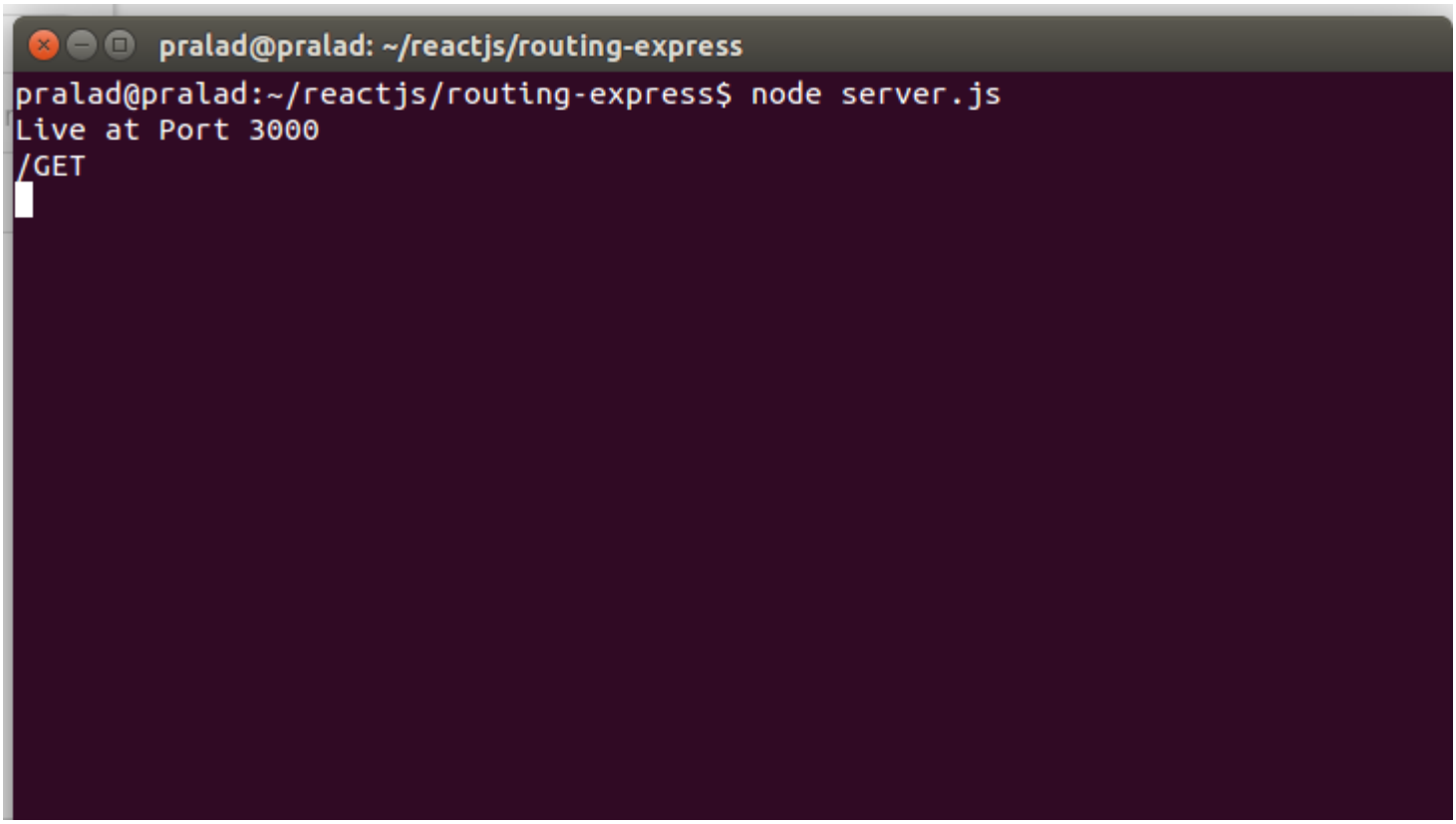
```
var express = require("express");
var app = express();

//Creating Router() object

var router = express.Router();

// Router middleware, mentioned it before defining routes.

router.use(function(req,res,next) {
  console.log("/" + req.method);
  next();
});

// Provide all routes here, this is for Home page.

router.get("/",function(req,res){
  res.json({"message" : "Hello World"});
});


app.use("/api",router);


app.listen(3000,function(){
  console.log("Live at Port 3000");
```

```
});
```

Ora se riavvii il server e hai fatto la richiesta a

```
http://localhost:3000/api/
```

Vedrai qualcosa di simile

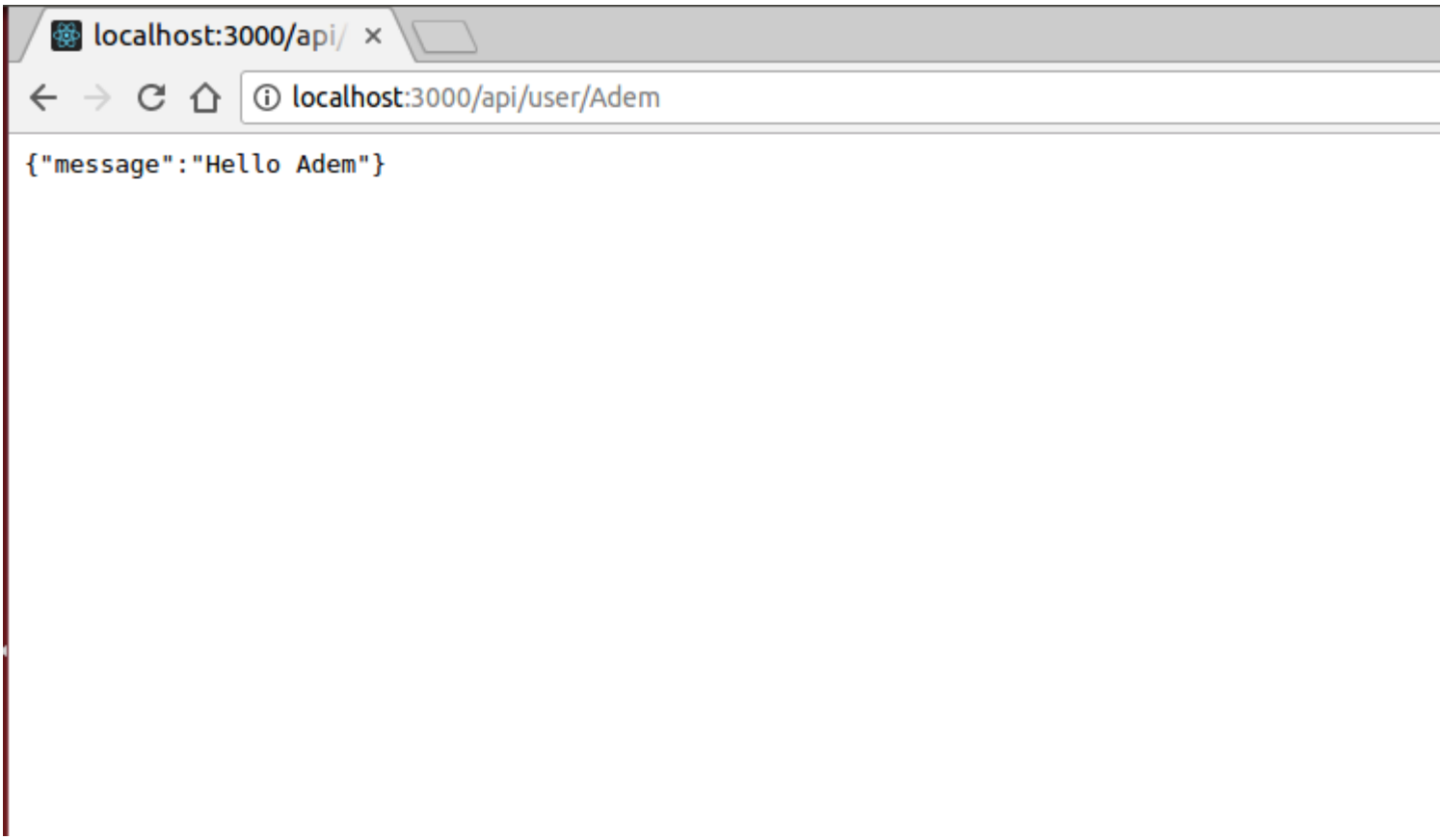


Accesso ai parametri nel routing

Puoi anche accedere al parametro dall'URL, come *http://example.com/api/:name/* . Quindi il parametro name può essere accesso. Aggiungi il seguente codice nel tuo server.js

```
router.get("/user/:id",function(req,res){
  res.json({"message" : "Hello "+req.params.id});
});
```

Ora riavvia il server e vai a [http: // localhost: 3000 / api / user / Adem] [4] , l'output sarà come



.

Leggi Routing NodeJs online: https://riptutorial.com/it/node-js/topic/9846/routing-nodejs

Capitolo 101: Sequelize.js

Examples

Installazione

Assicurati di aver prima installato Node.js e npm. Quindi installare sequelize.js con npm

```
npm install --save sequelize
```

Sarà inoltre necessario installare i moduli Node.js del database supportati. Devi solo installare quello che stai utilizzando

Per `MYSQL` e `Mariadb`

```
npm install --save mysql
```

Per `PostgreSQL`

```
npm install --save pg pg-hstore
```

Per `SQLite`

```
 npm install --save sqlite
```

Per `MSSQL`

```
npm install --save tedious
```

Una volta installato, puoi includere e creare una nuova istanza di Sequalize in questo modo.

Sintassi ES5

```
var Sequelize = require('sequelize');
var sequelize = new Sequelize('database', 'username', 'password');
```

Sintassi di Babele ES6 stage-0

```
import Sequelize from 'sequelize';
const sequelize = new Sequelize('database', 'username', 'password');
```

Ora hai un'istanza di sequelize disponibile. Potresti se ti senti così incline chiamarlo con un nome diverso come

```
var db = new Sequelize('database', 'username', 'password');
```

o

```
var database = new Sequelize('database', 'username', 'password');
```

quella parte è la tua prerogativa. Una volta installato, puoi utilizzarlo all'interno dell'applicazione come da documentazione dell'API http://docs.sequelizejs.com/en/v3/api/sequelize/

Il prossimo passo dopo l'installazione sarebbe quello di impostare il proprio modello

Definizione dei modelli

Ci sono due modi per definire i modelli in sequelize; con `sequelize.define(...)` o `sequelize.import(...)` . Entrambe le funzioni restituiscono un oggetto modello sequelizzato.

1. sequelize.define (modelName, attributes, [opzioni])

Questa è la strada da percorrere se si desidera definire tutti i modelli in un unico file o se si desidera avere un ulteriore controllo sulla definizione del modello.

```
/* Initialize Sequelize */
const config = {
    username: "database username",
    password: "database password",
    database: "database name",
    host: "database's host URL",
    dialect: "mysql" // Other options are postgres, sqlite, mariadb and mssql.
}
var Sequelize = require("sequelize");
var sequelize = new Sequelize(config);

/* Define Models */
sequelize.define("MyModel", {
    name: Sequelize.STRING,
    comment: Sequelize.TEXT,
    date: {
        type: Sequelize.DATE,
        allowNull: false
    }
});
```

Per la documentazione e altri esempi, consultare la documentazione di doclets o la documentazione di sequelize.com .

---

2. sequelize.import (path)

Se le definizioni del modello sono suddivise in un file per ciascuna, quindi l' `import` è tuo amico.

Nel file in cui si inizializza Sequelize, è necessario chiamare l'importazione in questo modo:

```
/* Initialize Sequelize */
// Check previous code snippet for initialization

/* Define Models */
sequelize.import("./models/my_model.js"); // The path could be relative or absolute
```

Quindi nei file di definizione del modello, il codice sarà simile a questo:

```
module.exports = function(sequelize, DataTypes) {
    return sequelize.define("MyModel", {
        name: DataTypes.STRING,
        comment: DataTypes.TEXT,
        date: {
            type: DataTypes.DATE,
            allowNull: false
        }
    });
};
```

Per ulteriori informazioni su come utilizzare l' `import` , controlla l' esempio esplicito di sequelize su GitHub .

Leggi Sequelize.js online: https://riptutorial.com/it/node-js/topic/7705/sequelize-js

Capitolo 102: Sfide di prestazione

Examples

Elaborazione di query a esecuzione prolungata con nodo

Poiché il nodo è a thread singolo, è necessario risolvere il problema in caso di calcoli a lunga esecuzione.

Nota: questo è l'esempio "pronto per l'esecuzione". Solo, non dimenticare di ottenere jQuery e installare i moduli richiesti.

Logica principale di questo esempio:

1. Il cliente invia una richiesta al server.
2. Il server avvia la routine in un'istanza del nodo separato e invia una risposta immediata con l'ID dell'attività correlata.
3. Il cliente invia continuamente assegni a un server per gli aggiornamenti di stato dell'ID attività specificato.

Struttura del progetto:

```
project
|   package.json
|   index.html
|
├──js
|      main.js
|      jquery-1.12.0.min.js
|
└──srv
    |    app.js
    ├── models
    |      task.js
    └── tasks
           data-processor.js
```

app.js:

```
var express     = require('express');
var app         = express();
var http        = require('http').Server(app);
var mongoose    = require('mongoose');
var bodyParser  = require('body-parser');

var childProcess= require('child_process');

var Task        = require('./models/task');

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

```
app.use(express.static(__dirname + '/../'));

app.get('/', function(request, response){
    response.render('index.html');
});

//route for the request itself
app.post('/long-running-request', function(request, response){
    //create new task item for status tracking
    var t = new Task({ status: 'Starting ...' });

    t.save(function(err, task){
        //create new instance of node for running separate task in another thread
        taskProcessor = childProcess.fork('./srv/tasks/data-processor.js');

        //process the messages comming from the task processor
        taskProcessor.on('message', function(msg){
            task.status = msg.status;
            task.save();
        }.bind(this));

        //remove previously openned node instance when we finished
        taskProcessor.on('close', function(msg){
            this.kill();
        });

        //send some params to our separate task
        var params = {
            message: 'Hello from main thread'
        };

        taskProcessor.send(params);
        response.status(200).json(task);
    });
});

//route to check is the request is finished the calculations
app.post('/is-ready', function(request, response){
    Task
        .findById(request.body.id)
        .exec(function(err, task){
            response.status(200).json(task);
        });
});

mongoose.connect('mongodb://localhost/test');
http.listen('1234');
```

task.js:

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

var taskSchema = mongoose.Schema({
    status: {
        type: String
    }
});

mongoose.model('Task', taskSchema);
```

```
module.exports  = mongoose.model('Task');
```

data-processor.js:

```
process.on('message', function(msg){
    init = function(){
        processData(msg.message);
    }.bind(this)();

    function processData(message){
        //send status update to the main app
        process.send({ status: 'We have started processing your data.' });

        //long calculations ..
        setTimeout(function(){
            process.send({ status: 'Done!' });

            //notify node, that we are done with this task
            process.disconnect();
        }, 5000);
    }
});

process.on('uncaughtException',function(err){
    console.log("Error happened: " + err.message + "\n" + err.stack + ".\n");
    console.log("Gracefully finish the routine.");
});
```

index.html:

```
<!DOCTYPE html>
<html>
    <head>
        <script src="./js/jquery-1.12.0.min.js"></script>
        <script src="./js/main.js"></script>
    </head>
    <body>
        <p>Example of processing long-running node requests.</p>
        <button id="go" type="button">Run</button>

        <br />

        <p>Log:</p>
        <textarea id="log" rows="20" cols="50"></textarea>
    </body>
</html>
```

main.js:

```
$(document).on('ready', function(){

    $('#go').on('click', function(e){
        //clear log
        $("#log").val('');

        $.post("/long-running-request", {some_params: 'params' })
            .done(function(task){
```

```
                $("#log").val( $("#log").val() + '\n' + task.status);

                //function for tracking the status of the task
                function updateStatus(){
                    $.post("/is-ready", {id: task._id })
                        .done(function(response){
                            $("#log").val( $("#log").val() + '\n' + response.status);

                            if(response.status != 'Done!'){
                                checkTaskTimeout = setTimeout(updateStatus, 500);
                            }
                        });
                }

                //start checking the task
                var checkTaskTimeout = setTimeout(updateStatus, 100);
            });
    });
});
```

package.json:

```
{
  "name": "nodeProcessor",
  "dependencies": {
    "body-parser": "^1.15.2",
    "express": "^4.14.0",
    "html": "0.0.10",
    "mongoose": "^4.5.5"
  }
}
```

Disclaimer: questo esempio ha lo scopo di darti un'idea di base. Per usarlo nell'ambiente di produzione, ha bisogno di miglioramenti.

Leggi Sfide di prestazione online: https://riptutorial.com/it/node-js/topic/6325/sfide-di-prestazione

Capitolo 103: Socket TCP

Examples

Un semplice server TCP

```
// Include Nodejs' net module.
const Net = require('net');
// The port on which the server is listening.
const port = 8080;

// Use net.createServer() in your code. This is just for illustration purpose.
// Create a new TCP server.
const server = new Net.Server();
// The server listens to a socket for a client to make a connection request.
// Think of a socket as an end point.
server.listen(port, function() {
    console.log(`Server listening for connection requests on socket localhost:${port}`.);
});

// When a client requests a connection with the server, the server creates a new
// socket dedicated to that client.
server.on('connection', function(socket) {
    console.log('A new connection has been established.');

    // Now that a TCP connection has been established, the server can send data to
    // the client by writing to its socket.
    socket.write('Hello, client.');

    // The server can also receive data from the client by reading from its socket.
    socket.on('data', function(chunk) {
        console.log(`Data received from client: ${chunk.toString()`.});
    });

    // When the client requests to end the TCP connection with the server, the server
    // ends the connection.
    socket.on('end', function() {
        console.log('Closing connection with the client');
    });

    // Don't forget to catch error, for your own sake.
    socket.on('error', function(err) {
        console.log(`Error: ${err}`);
    });
});
```

Un semplice client TCP

```
// Include Nodejs' net module.
const Net = require('net');
// The port number and hostname of the server.
const port = 8080;
const host = 'localhost';

// Create a new TCP client.
```

```
const client = new Net.Socket();
// Send a connection request to the server.
client.connect({ port: port, host: host }), function() {
    // If there is no error, the server has accepted the request and created a new
    // socket dedicated to us.
    console.log('TCP connection established with the server.');

    // The client can now send data to the server by writing to its socket.
    client.write('Hello, server.');
});

// The client can also receive data from the server by reading from its socket.
client.on('data', function(chunk) {
    console.log(`Data received from the server: ${chunk.toString()}.`);

    // Request an end to the connection after the data has been received.
    client.end();
});

client.on('end', function() {
    console.log('Requested an end to the TCP connection');
});
```

Leggi Socket TCP online: https://riptutorial.com/it/node-js/topic/6545/socket-tcp

Capitolo 104: Storia di Nodejs

introduzione

Qui discuteremo della storia di Node.js, delle informazioni sulla versione e del suo stato attuale.

Examples

Eventi chiave in ogni anno

2009

- 3 marzo: il progetto è stato nominato "nodo"
- 1 ottobre: prima anteprima molto presto di npm, il pacchetto Node manager
- 8 novembre: Ryan Dahl's (Creator of Node.js) Original Node.js Talk al JSConf 2009

2010

- Express: un framework di sviluppo web Node.js
- Socket.io versione iniziale
- 28 aprile: supporto Experimental Node.js su Heroku
- 28 luglio: Google Tech Talk di Ryan Dahl su Node.js
- 20 agosto: rilasciato Node.js 0.2.0

2011

- 31 marzo: Guida Node.js
- 1 maggio: npm 1.0: rilasciato
- 1 maggio: AMA di Ryan Dahl su Reddit
- 10 luglio: Il Node Beginner Book, un'introduzione a Node.js, è completato .
    - Un tutorial completo su Node.js per principianti.
- 16 agosto: LinkedIn utilizza Node.js
    - LinkedIn ha lanciato la sua app mobile completamente rinnovata con nuove funzionalità e nuove parti sotto il cofano.
- 5 ottobre: Ryan Dahl parla della storia di Node.js e del motivo per cui l'ha creata
- 5 dicembre: Node.js in produzione presso Uber
    - Il responsabile della progettazione di Uber Curtis Chambers spiega perché la sua azienda ha completamente riprogettato la propria applicazione utilizzando Node.js per aumentare l'efficienza e migliorare l'esperienza dei partner e dei clienti.

2012

- 30 gennaio: il creatore di Node.js Ryan Dahl si allontana quotidianamente da Node
- 25 giugno: Node.js v0.8.0 [stable] è fuori
- 20 dicembre: Hapi, viene rilasciato un framework Node.js

2013

- 30 aprile: Lo stack MEAN: MongoDB, ExpressJS, AngularJS e Node.js
- 17 maggio: Come abbiamo costruito la prima applicazione Node.js di eBay
- 15 novembre: PayPal rilascia Kraken, un framework Node.js
- 22 novembre: perdita di memoria Node.js a Walmart
    - I laboratori di Eran Hammer of Wal-Mart sono venuti al nucleo di Node.js per lamentarsi di una perdita di memoria che stava rintracciando da mesi.
- 19 dicembre: Koa - Web framework per Node.js

2014

- 15 gennaio: TJ Fontaine rileva il progetto Node
- 23 ottobre: Node.js Advisory Board
    - Joyent e diversi membri della comunità Node.js hanno annunciato una proposta per un Advisory Board Node.js come passo successivo verso un modello di governance completamente aperto per il progetto open source Node.js.
- 19 novembre: Node.js in Flame Graphs - Netflix
- 28 novembre: IO.js - I / O evento per V8 Javascript

2015

Q1

- 14 gennaio: IO.js 1.0.0
- 10th Febraury: Joyent si muove per stabilire la fondazione Node.js
    - Joyent, IBM, Microsoft, PayPal, Fidelity, SAP e The Linux Foundation uniscono le forze per supportare la community Node.js con Neutral e Open Governance
- 27 febbraio: IO.js e proposta di riconciliazione Node.js

Q2

- 14 aprile: npm Private Modules
- 28 maggio: il leader del Nodo TJ Fontaine si dimette e lascia Joyent
- 13 maggio: Node.js e io.js si uniscono sotto la Node Foundation

Q3

- 2 agosto: Trace - Monitoraggio e debug delle prestazioni di Node.js
    - Trace è uno strumento di monitoraggio dei microservizi visualizzato che consente di ottenere tutti i parametri necessari quando si utilizzano i microservizi.
- 13 agosto: 4.0 è il nuovo 1.0

Q4

- 12 ottobre: Nodo v4.2.0, prima versione del supporto a lungo termine
- 8 dicembre: Apigee, RisingStack e Yahoo si uniscono alla Node.js Foundation
- 8 e 9 dicembre: Node Interactive
    - La prima conferenza annuale Node.js della Node.js Foundation

2016

Q1

- 10 febbraio: Express diventa un progetto incubato
- 23 marzo: incidente a sinistra
- 29 marzo: Google Cloud Platform si unisce alla Node.js Foundation

Q2

- 26 aprile: npm ha 210.000 utenti

Q3

- 18 luglio: CJ Silverio diventa CTO di npm
- 1 ° agosto: Trace, la soluzione di debug di Node.js diventa generalmente disponibile
- 15 settembre: il primo Node Interactive in Europa

Q4

- 11 ottobre: viene rilasciato il gestore del pacchetto di filati
- 18 ottobre: Node.js 6 diventa la versione LTS

Riferimento

1. "Cronologia di Node.js su una linea temporale" [Online]. Disponibile: [https://blog.risingstack.com/history-of-node-js]

Leggi Storia di Nodejs online: https://riptutorial.com/it/node-js/topic/8653/storia-di-nodejs

Capitolo 105: Struttura del progetto

introduzione

La struttura del progetto nodejs è influenzata dalle preferenze personali, dall'architettura del progetto e dalla strategia di iniezione del modulo in uso. Inoltre, l'arco basato su eventi 'che utilizza il meccanismo di istanziazione dei moduli dinamici. Per avere una struttura MVC è imperativo separare il lato server e il codice sorgente lato client, poiché il codice lato client verrà probabilmente ridotto a icona e inviato al browser ed è pubblico nella sua natura di base. E il lato server o il back-end forniranno API per eseguire operazioni CRUD

Osservazioni

Il progetto in alto utilizza i moduli browserify e vue.js come librerie view e minification dell'applicazione base. Quindi la struttura del progetto può cambiare minuziosamente in base al framework mvc che usi, ad esempio La directory di compilazione in pubblico dovrà contenere tutto il codice mvc. Puoi avere un compito che fa questo per te.

Examples

Una semplice applicazione nodejs con MVC e API

- La prima distinzione principale è tra le directory generate dinamicamente che verranno utilizzate per le directory di hosting e di origine.
- Le directory di origine avranno un file di configurazione o una cartella a seconda della quantità di configurazione che potresti avere. Ciò include la configurazione dell'ambiente e la configurazione della business logic che è possibile scegliere di inserire nella directory di configurazione.

```
|-- Config
       |-- config.json
       |-- appConfig
           |-- pets.config
           |-- payment.config
```

- Ora le directory più vitali in cui distinguiamo tra lato server / back-end e moduli front-end. Il *server di* 2 directory e *webapp* rappresentano rispettivamente il backend e il frontend che possiamo scegliere di inserire all'interno di una directory di origine. *src* .

  > Puoi scegliere nomi diversi per scelta personale per server o webapp a seconda di cosa ha senso per te. Assicurati di non volerlo fare troppo a lungo o troppo complesso come nella struttura interna del progetto finale.

- All'interno della directory del *server* puoi avere il controller, l'app.js / index.js che sarà il tuo file mainjs principale e il punto di partenza. La dir del server. può anche avere la *directory*

dto che contiene tutti gli oggetti di trasferimento dati che saranno usd dai controller API.

```
|-- server
     |-- dto
         |-- pet.js
         |-- payment.js
     |-- controller
         |-- PetsController.js
         |-- PaymentController.js
     |-- App.js
```

- La directory webapp può essere divisa in due parti principali *public* e *mvc* , questo è di nuovo influenzato dalla strategia di build che si desidera utilizzare. Stiamo usando browserfiy per costruire la parte MVC di webapp e minimizzare il contenuto della directory *mvc* semplicemente messo.

  | - webapp | - public | - mvc

- Ora la directory pubblica può contenere tutte le risorse statiche, le immagini, i css (anche i file saas) e, soprattutto, i file HTML.

```
|-- public
    |-- build  // will contianed minified scripts(mvc)
    |-- images
        |-- mouse.jpg
        |-- cat.jpg
    |-- styles
        |-- style.css
    |-- views
        |-- petStore.html
        |-- paymentGateway.html
        |-- header.html
        |-- footer.html
    |-- index.html
```

- La directory *mvc* conterrà la logica front-end inclusi i *modelli* , i *controller di visualizzazione* e qualsiasi altro modulo di *utilità* che potrebbe essere necessario come parte dell'interfaccia utente. Anche index.js o shell.js, qualunque sia la tua suite, fa parte di questa directory.

```
|-- mvc
    |-- controllers
        |-- Dashborad.js
        |-- Help.js
        |-- Login.js
    |-- utils
    |-- index.js
```

Quindi, in conclusione, l'intera struttura del progetto apparirà come segue. E una semplice attività di build come *gulp browserify* ridurrà gli script mvc e pubblicherà nella directory *pubblica* . Possiamo quindi fornire questa directory pubblica come risorsa statica tramite ***express.use (satic ('public'))*** api.

```
    |-- node_modules
```

```
|-- src
    |-- server
        |-- controller
        |-- App.js   // node app
    |-- webapp
        |-- public
            |-- styles
            |-- images
            |-- index.html
        |-- mvc
            |-- controller
            |-- shell.js  // mvc shell
|-- config
|-- Readme.md
|-- .gitignore
|-- package.json
```

Leggi Struttura del progetto online: https://riptutorial.com/it/node-js/topic/9935/struttura-del-progetto

Capitolo 106: Upload di file

Examples

Caricamento file singolo con multer

Ricordati di

- crea una cartella per il caricamento (`uploads` in esempio).
- installare `npm i -S multer`

`server.js` :

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

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

var storage =   multer.diskStorage({
  destination: function (req, file, callback) {
    fs.mkdir('./uploads', function(err) {
        if(err) {
            console.log(err.stack)
        } else {
            callback(null, './uploads');
        }
    })
  },
  filename: function (req, file, callback) {
    callback(null, file.fieldname + '-' + Date.now());
  }
});

app.post('/api/file',function(req,res){
    var upload = multer({ storage : storage}).single('userFile');
    upload(req,res,function(err) {
        if(err) {
            return res.end("Error uploading file.");
        }
        res.end("File is uploaded");
    });
});

app.listen(3000,function(){
    console.log("Working on port 3000");
});
```

`index.html` :

```
<form id        =  "uploadForm"
```

```
     enctype   =  "multipart/form-data"
     action    =  "/api/file"
     method    =  "post"
>
<input type="file" name="userFile" />
<input type="submit" value="Upload File" name="submit">
</form>
```

Nota:

Per caricare file con estensione è possibile utilizzare la libreria integrata del percorso Node.js

Per questo basta richiedere il `path` file `server.js` :

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

e cambia:

```
callback(null, file.fieldname + '-' + Date.now());
```

aggiungendo un'estensione di file nel modo seguente:

```
callback(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname));
```

Come filtrare il caricamento per estensione:

In questo esempio, vedi come caricare i file per consentire solo determinate estensioni.

Ad esempio solo le estensioni delle immagini. Basta aggiungere a `var upload = multer({ storage : storage}).single('userFile');` condizione file filtro

```
var upload = multer({
    storage: storage,
    fileFilter: function (req, file, callback) {
        var ext = path.extname(file.originalname);
        if(ext !== '.png' && ext !== '.jpg' && ext !== '.gif' && ext !== '.jpeg') {
            return callback(new Error('Only images are allowed'))
        }
        callback(null, true)
    }
}).single('userFile');
```

Ora puoi caricare solo file immagine con estensioni `png` , `jpg` , `gif` o `jpeg`

Usando il modulo formidabile

Installa il modulo e leggi i **documenti**

```
npm i formidable@latest
```

Esempio di server sulla porta 8080

```
var formidable = require('formidable'),
    http = require('http'),
    util = require('util');

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // parse a file upload
    var form = new formidable.IncomingForm();

    form.parse(req, function(err, fields, files) {
      if (err)
          do-smth; // process error

      // Copy file from temporary place
      // var fs = require('fs');
      // fs.rename(file.path, <targetPath>, function (err) { ... });

      // Send result on client
      res.writeHead(200, {'content-type': 'text/plain'});
      res.write('received upload:\n\n');
      res.end(util.inspect({fields: fields, files: files}));
    });

    return;
  }

  // show a file upload form
  res.writeHead(200, {'content-type': 'text/html'});
  res.end(
    '<form action="/upload" enctype="multipart/form-data" method="post">'+
    '<input type="text" name="title"><br>'+
    '<input type="file" name="upload" multiple="multiple"><br>'+
    '<input type="submit" value="Upload">'+
    '</form>'
  );
}).listen(8080);
```

Leggi Upload di file online: https://riptutorial.com/it/node-js/topic/4080/upload-di-file

Capitolo 107: Usa casi di Node.js

Examples

Server HTTP

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

console.log('Starting server...');
var config = {
    port: 80,
    contentType: 'application/json; charset=utf-8'
};
// JSON-API server on port 80

var server = http.createServer();
server.listen(config.port);
server.on('error', (err) => {
    if (err.code == 'EADDRINUSE') console.error('Port '+ config.port +' is already in use');
    else console.error(err.message);
});
server.on('request', (request, res) => {
    var remoteAddress = request.headers['x-forwarded-for'] ||
request.connection.remoteAddress; // Client address
    console.log(remoteAddress +' '+ request.method +' '+ request.url);

    var out = {};
    // Here you can change output according to `request.url`
    out.test = request.url;
    res.writeHead(200, {
        'Content-Type': config.contentType
    });
    res.end(JSON.stringify(out));
});
server.on('listening', () => {
    c.info('Server is available: http://localhost:'+ config.port);
});
```

Console con prompt dei comandi

```
const process = require('process');
const rl = require('readline').createInterface(process.stdin, process.stdout);

rl.pause();
console.log('Something long is happening here...');

var cliConfig = {
    promptPrefix: ' > '
}

/*
    Commands recognition
    BEGIN
*/
var commands = {
```

```
    eval: function(arg) { // Try typing in console: eval 2 * 10 ^ 3 + 2 ^ 4
        arg = arg.join(' ');
        try { console.log(eval(arg)); }
        catch (e) { console.log(e); }
    },
    exit: function(arg) {
        process.exit();
    }
};
rl.on('line', (str) => {
    rl.pause();
    var arg = str.trim().match(/([^"]+)|("(?:[^"\\]|\\.)+")/g); // Applying regular expression
for removing all spaces except for what between double quotes:
http://stackoverflow.com/a/14540319/2396907
    if (arg) {
        for (let n in arg) {
            arg[n] = arg[n].replace(/^\"|\"$/g, '');
        }
        var commandName = arg[0];
        var command = commands[commandName];
        if (command) {
            arg.shift();
            command(arg);
        }
        else console.log('Command "'+ commandName +'" doesn\'t exist');
    }
    rl.prompt();
});
/*
    END OF
    Commands recognition
*/

rl.setPrompt(cliConfig.promptPrefix);
rl.prompt();
```

Leggi Usa casi di Node.js online: https://riptutorial.com/it/node-js/topic/7703/usa-casi-di-node-js

Capitolo 108: Usando i flussi

Parametri

Parametro	Definizione
Stream leggibile	tipo di flusso da cui è possibile leggere i dati
Flusso scrivibile	tipo di flusso in cui i dati possono essere scritti
Duplex Stream	tipo di stream che è sia leggibile che scrivibile
Trasforma stream	tipo di flusso duplex in grado di trasformare i dati mentre vengono letti e quindi scritti

Examples

Leggi i dati da TextFile con flussi

L'I / O nel nodo è asincrono, quindi l'interazione con il disco e la rete implica il passaggio di callback alle funzioni. Potresti essere tentato di scrivere il codice che serve un file dal disco in questo modo:

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

var server = http.createServer(function (req, res) {
    fs.readFile(__dirname + '/data.txt', function (err, data) {
        res.end(data);
    });
});
server.listen(8000);
```

Questo codice funziona ma è ingombrante e memorizza l'intero file data.txt in memoria per ogni richiesta prima di scrivere il risultato ai client. Se data.txt è molto grande, il tuo programma potrebbe iniziare a consumare molta memoria poiché serve molti utenti contemporaneamente, in particolare per gli utenti con connessioni lente.

Anche l'esperienza utente è scadente perché gli utenti dovranno attendere che l'intero file venga memorizzato in memoria sul server prima che possano iniziare a ricevere qualsiasi contenuto.

Fortunatamente entrambi gli argomenti (req, res) sono stream, il che significa che possiamo scrivere questo in un modo molto migliore usando fs.createReadStream () invece di fs.readFile ():

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

```
var server = http.createServer(function (req, res) {
    var stream = fs.createReadStream(__dirname + '/data.txt');
    stream.pipe(res);
});
server.listen(8000);
```

Qui .pipe () si occupa di ascoltare gli eventi "data" e "end" da fs.createReadStream (). Questo codice non è solo più pulito, ma ora il file data.txt verrà scritto ai client un chunk alla volta immediatamente quando vengono ricevuti dal disco.

Flussi di tubazioni

Gli stream leggibili possono essere "convogliati" o collegati a flussi scrivibili. Ciò rende il flusso di dati dal flusso di origine al flusso di destinazione senza molti sforzi.

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

var readable = fs.createReadStream('file1.txt')
var writable = fs.createWriteStream('file2.txt')

readable.pipe(writable) // returns writable
```

Quando i flussi scrivibili sono anche flussi leggibili, vale a dire quando sono flussi *duplex* , è possibile continuare a collegarli ad altri flussi scrivibili.

```
var zlib = require('zlib')

fs.createReadStream('style.css')
  .pipe(zlib.createGzip()) // The returned object, zlib.Gzip, is a duplex stream.
  .pipe(fs.createWriteStream('style.css.gz')
```

Anche i flussi leggibili possono essere inviati in più flussi.

```
var readable = fs.createReadStream('source.css')
readable.pipe(zlib.createGzip()).pipe(fs.createWriteStream('output.css.gz'))
readable.pipe(fs.createWriteStream('output.css')
```

Si noti che è necessario reindirizzare i flussi di output in modo sincrono (allo stesso tempo) prima che i dati "scorrano". In caso contrario, i dati incompleti potrebbero essere trasmessi in streaming.

Si noti inoltre che gli oggetti stream possono emettere eventi di `error` ; assicurati di gestire responsabilmente questi eventi su *ogni* stream, se necessario:

```
var readable = fs.createReadStream('file3.txt')
var writable = fs.createWriteStream('file4.txt')
readable.pipe(writable)
readable.on('error', console.error)
writable.on('error', console.error)
```

Crea il tuo stream leggibile / scrivibile

Vedremo oggetti stream restituiti da moduli come fs etc, ma se vogliamo creare il nostro oggetto streamable.

Per creare un oggetto Stream è necessario utilizzare il modulo stream fornito da NodeJs

```
    var fs = require("fs");
    var stream = require("stream").Writable;

    /*
     *  Implementing the write function in writable stream class.
     *  This is the function which will be used when other stream is piped into this
     *  writable stream.
     */
    stream.prototype._write = function(chunk, data){
        console.log(data);
    }

    var customStream = new stream();

    fs.createReadStream("am1.js").pipe(customStream);
```

Questo ci darà il nostro flusso scrivibile personalizzato. possiamo implementare qualsiasi cosa all'interno della funzione *_write* . Il metodo sopra funziona nella versione 4.xx di NodeJs ma in NodeJs 6.x **ES6 ha** introdotto le classi, pertanto la sintassi è stata modificata. Di seguito è riportato il codice per la versione 6.x di NodeJs

```
    const Writable = require('stream').Writable;

    class MyWritable extends Writable {
      constructor(options) {
        super(options);
      }

      _write(chunk, encoding, callback) {
        console.log(chunk);
      }
    }
```

Perché i flussi?

Esaminiamo i seguenti due esempi per leggere il contenuto di un file:

Il primo, che utilizza un metodo asincrono per leggere un file e fornisce una funzione di callback che viene chiamata una volta che il file è stato completamente letto nella memoria:

```
fs.readFile(`${__dirname}/utils.js`, (err, data) => {
  if (err) {
    handleError(err);
  } else {
    console.log(data.toString());
  }
})
```

E il secondo, che utilizza i `streams` per leggere il contenuto del file, pezzo per pezzo:

```
var fileStream = fs.createReadStream(`${__dirname}/file`);
var fileContent = '';
fileStream.on('data', data => {
  fileContent += data.toString();
})

fileStream.on('end', () => {
  console.log(fileContent);
})

fileStream.on('error', err => {
  handleError(err)
})
```

Vale la pena ricordare che entrambi gli esempi fanno **esattamente la stessa cosa** . Qual è la differenza allora?

- Il primo è più corto e sembra più elegante
- Il secondo consente di eseguire qualche elaborazione sul file **mentre** viene letto (!)

Quando i file con cui si gestiscono sono piccoli, non c'è alcun effetto reale quando si usano gli `streams` , ma cosa succede quando il file è grande? (così grande che ci vogliono 10 secondi per leggerlo in memoria)

Senza `streams` ti aspetteresti, senza fare assolutamente nulla (a meno che il tuo processo non faccia altro), fino a quando non passeranno i 10 secondi e il file sarà **completamente letto** , e solo allora potrai iniziare l'elaborazione del file.

Con i `streams` , si ottiene il contenuto del file pezzo per pezzo, **proprio quando sono disponibili** , e ciò consente di elaborare il file **mentre** viene letto.

L'esempio precedente non illustra come gli `streams` possono essere utilizzati per lavori che non possono essere eseguiti quando si fa il callback, quindi guardiamo un altro esempio:

Vorrei scaricare un file `gzip` , decomprimerlo e salvarne il contenuto sul disco. Dato l' `url` del file, questo è ciò che è necessario fare:

- Scarica il file
- Decomprimere il file
- Salvalo su disco

Ecco un [file piccolo] [1], che è memorizzato nella mia memoria `S3` . Il seguente codice fa quanto sopra nel modo callback.

```
var startTime = Date.now()
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}, (err, data) => {
  // here, the whole file was downloaded

  zlib.gunzip(data.Body, (err, data) => {
    // here, the whole file was unzipped

    fs.writeFile(`${__dirname}/tweets.json`, data, err => {
```

```
      if (err) console.error(err)

      // here, the whole file was written to disk
      var endTime = Date.now()
      console.log(`${endTime - startTime} milliseconds`) // 1339 milliseconds
    })
  })
})

// 1339 milliseconds
```

Ecco come appare usando i `streams` :

```
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}).createReadStream()
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream(`${__dirname}/tweets.json`));

// 1204 milliseconds
```

Sì, non è più veloce quando si ha a che fare con file di piccole dimensioni: il file testato pesa `80KB` . Provando questo su un file più grande, `71MB` gzipped (`382MB` decompresso), mostra che la versione degli `streams` è molto più veloce

- Sono stati necessari 20925 millisecondi per scaricare `71MB` , decomprimerlo e quindi scrivere `382MB` su disco, **utilizzando la modalità callback** .
- In confronto, ci sono voluti 13434 millisecondi per fare lo stesso quando si utilizza la versione di `streams` (35% più veloce, per un file non così grande)

Leggi Usando i flussi online: https://riptutorial.com/it/node-js/topic/2974/usando-i-flussi

Capitolo 109: Usare Browserfiy per risolvere l'errore 'richiesto' con i browser

Examples

Esempio: file.js

In questo esempio abbiamo un file chiamato **file.js.**

Supponiamo di dover analizzare un URL utilizzando il modulo querystring di JavaScript e NodeJS.

Per fare questo, tutto ciò che devi fare è inserire la seguente dichiarazione nel tuo file:

```
const querystring = require('querystring');
var ref = querystring.parse("foo=bar&abc=xyz&abc=123");
```

Cosa sta facendo questo frammento?

Bene, per prima cosa, creiamo un modulo querystring che fornisce utility per l'analisi e la formattazione delle stringhe di query URL. Si può accedere usando:

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

Quindi, analizziamo un URL usando il metodo .parse (). Analizza una stringa di query URL (str) in un insieme di coppie chiave e valore.

Ad esempio, la stringa di query `'foo=bar&abc=xyz&abc=123'` viene analizzata in:

```
{  foo: 'bar',  abc: ['xyz', '123']   }
```

Sfortunatamente, i browser non hanno il metodo *richiesto* definito, ma lo fa Node.js.

Installa Browserfy

Con Browserify puoi scrivere il codice che usa *richiede* nello stesso modo in cui lo useresti nel nodo. Quindi, come risolvi questo? È semplice.

1. Primo nodo di installazione, fornito con npm. Quindi fa:

   npm install -g **browserify**

2. Cambia nella directory in cui si trova il file.js e Installa il nostro modulo *querystring* con npm:

npm install **querystring**

Nota: se non si modifica la directory specifica, il comando avrà esito negativo poiché non riesce a trovare il file che contiene il modulo.

3. Ora raggruppa in modo ricorsivo tutti i moduli necessari a partire da file.js in un unico file chiamato bundle.js (o quello che ti piace **chiamarlo**) con il **comando browserify** :

   browserify file.js -o bundle.js

Browserify analizza l'Abstract Syntax Tree per le chiamate *require ()* per attraversare l'intero grafico delle dipendenze del tuo

4. Infine scendi un singolo tag nel tuo html e il gioco è fatto!

```
<script src="bundle.js"></script>
```

Quello che succede è che ottieni una combinazione del tuo vecchio file .js (**file.js**) e del tuo file **bundle.js** appena creato. Questi due file sono uniti in un singolo file.

Importante

Si prega di tenere presente che se si desidera apportare modifiche al file.js e non influenzerà il comportamento del programma. **Le tue modifiche diventeranno effettive solo se modifichi il bundle.js appena creato**

Cosa significa?

Ciò significa che se si desidera modificare **file.js** per qualsiasi motivo, le modifiche non avranno alcun effetto. Devi davvero modificare **bundle.js** poiché è un'unione di **bundle.js** e **file.js.**

Leggi Usare Browserfiy per risolvere l'errore 'richiesto' con i browser online: https://riptutorial.com/it/node-js/topic/7123/usare-browserfiy-per-risolvere-l-errore--richiesto--con-i-browser

Capitolo 110: Utilizzando WebSocket con Node.JS

Examples

Installazione di WebSocket

Ci sono alcuni modi per installare WebSocket nel tuo progetto. Ecco alcuni esempi:

```
npm install --save ws
```

o all'interno del pacchetto package.json utilizzando:

```
"dependencies": {
  "ws": "*"
},
```

Aggiunta di WebSocket al tuo file

Per aggiungere ws al tuo file usa semplicemente:

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

Utilizzo dei server WebSocket e WebSocket

Per aprire un nuovo WebSocket, è sufficiente aggiungere qualcosa come:

```
var WebSocket = require("ws");
var ws = new WebSocket("ws://host:8080/OptionalPathName);
// Continue on with your code...
```

O per aprire un server, usa:

```
var WebSocketServer = require("ws").Server;
var ws = new WebSocketServer({port: 8080, path: "OptionalPathName"});
```

Un esempio di server WebSocket semplice

```
var WebSocketServer = require('ws').Server
, wss = new WebSocketServer({ port: 8080 }); // If you want to add a path as well, use path:
"PathName"

wss.on('connection', function connection(ws) {
  ws.on('message', function incoming(message) {
    console.log('received: %s', message);
  });
```

```
  ws.send('something');
});
```

Leggi Utilizzando WebSocket con Node.JS online: https://riptutorial.com/it/node-js/topic/6106/utilizzando-websocket-con-node-js

Capitolo 111: Utilizzo di IISNode per ospitare le app Web Node.js in IIS

Osservazioni

Directory virtuale / Applicazione nidificata con le trappole delle viste

Se si utilizzerà Express per eseguire il rendering delle viste utilizzando un motore di visualizzazione, sarà necessario passare il valore `virtualDirPath` alle visualizzazioni

```
`res.render('index', { virtualDirPath: virtualDirPath });`
```

La ragione per fare ciò è di rendere i tuoi collegamenti ipertestuali ad altre viste host dall'app e dai percorsi delle risorse statiche per sapere dove è ospitato il sito senza dover modificare tutte le viste dopo la distribuzione. Questa è una delle più fastidiose e noiose trappole dell'uso di Directory Virtuali con IISNode.

versioni

Tutti gli esempi sopra funzionano con

- Express v4.x
- IIS 7.x / 8.x
- Socket.io v1.3.x o successivo

Examples

Iniziare

IISNode consente alle app Web di Node.js di essere ospitate su IIS 7/8 proprio come farebbe un'applicazione .NET. Ovviamente, è possibile ospitare autonomamente il processo `node.exe` su Windows, ma perché farlo basta quando si esegue l'app in IIS.

IISNode gestirà il ridimensionamento su più core, la gestione dei processi di `node.exe` e il riciclo automatico dell'applicazione IIS ogni volta che la tua app viene aggiornata, solo per citarne alcuni dei suoi vantaggi .

Requisiti

IISNode ha alcuni requisiti prima di poter ospitare l'app Node.js in IIS.

1. Node.js deve essere installato sull'host IIS, a 32 o 64 bit, o sono supportati.
2. IISNode installato x86 o x64 , questo dovrebbe corrispondere al testimone del tuo host IIS.
3. Il modulo URL-Rewrite di Microsoft per IIS installato sul tuo host IIS.
    - Questa è la chiave, altrimenti le richieste alla tua app Node.js non funzioneranno come previsto.
4. Un `Web.config` nella cartella principale della tua app Node.js.
5. Configurazione IISNode tramite un file `iisnode.yml` o un elemento `<iisnode>` all'interno di `Web.config` .

Esempio di base Hello World utilizzando Express

Per ottenere questo esempio, è necessario creare un'applicazione IIS 7/8 sul proprio host IIS e aggiungere la directory contenente l'app Web Node.js come directory fisica. Assicurarsi che l'identità del pool di applicazioni / applicazioni possa accedere all'installazione di Node.js. Questo esempio utilizza l'installazione a 64 bit di Node.js.

Struttura del progetto

Questa è la struttura di progetto di base di un'app Web IISNode / Node.js. Sembra quasi identico a qualsiasi Web App non IISNode tranne per l'aggiunta di `Web.config` .

```
- /app_root
  - package.json
  - server.js
  - Web.config
```

server.js - Applicazione Express

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

// We need to get the port that IISNode passes into us
// using the PORT environment variable, if it isn't set use a default value
const port = process.env.PORT || 3000;

// Setup a route at the index of our app
server.get('/', (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});
```

Configurazione e Web.config

`Web.config` è simile a qualsiasi altro IIS `Web.config` tranne che le seguenti due cose devono essere presenti, URL `<rewrite><rules>` e un `<handler>` IISNode `<handler>` . Entrambi questi elementi sono figli dell'elemento `<system.webServer>` .

Configurazione

È possibile configurare IISNode utilizzando un file `iisnode.yml` o aggiungendo l'elemento `<iisnode>` come figlio di `<system.webServer>` nel proprio `Web.config` . Entrambe queste configurazioni possono essere utilizzate in combinazione tra loro, tuttavia, in questo caso, `Web.config` dovrà specificare il file `iisnode.yml` **E** qualsiasi conflitto di configurazione verrà preso dal file `iisnode.yml` . Questa sovrascrittura della configurazione non può avvenire al contrario.

IISNode Handler

Per fare in modo che IIS sappia che `server.js` contiene la nostra app Web Node.js, dobbiamo dirlo esplicitamente. Possiamo farlo aggiungendo IISNode `<handler>` `<handlers>` all'elemento `<handlers>` .

```
<handlers>
  <add name="iisnode" path="server.js" verb="*" modules="iisnode"/>
</handlers>
```

Regole di riscrittura degli URL

La parte finale della configurazione sta assicurando che il traffico destinato alla nostra app Node.js in arrivo nella nostra istanza IIS venga indirizzato a IISNode. Senza le regole di riscrittura degli URL, dovremmo visitare la nostra app andando su `http://<host>/server.js` e, peggio ancora, quando provi a richiedere una risorsa fornita da `server.js` otterrai un `404` . Questo è il motivo per cui la riscrittura degli URL è necessaria per le app Web IISNode.

```
<rewrite>
    <rules>
        <!-- First we consider whether the incoming URL matches a physical file in the /public
folder -->
        <rule name="StaticContent" patternSyntax="Wildcard">
            <action type="Rewrite" url="public/{R:0}" logRewrittenUrl="true"/>
            <conditions>
                <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true"/>
            </conditions>
            <match url="*.*"/>
        </rule>

        <!-- All other URLs are mapped to the Node.js application entry point -->
        <rule name="DynamicContent">
            <conditions>
                <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="True"/>
            </conditions>
            <action type="Rewrite" url="server.js"/>
        </rule>
    </rules>
</rewrite>
```

Questo è un file `Web.config` funzionante per questo esempio , installazione per un'installazione Node.js a 64 bit.

È tutto, ora visita il tuo sito IIS e vedi come funziona l'applicazione Node.js.

Utilizzando una directory virtuale IIS o un'applicazione nidificata tramite

L'utilizzo di una directory virtuale o di un'applicazione nidificata in IIS è uno scenario comune e molto probabilmente di cui si vorrà approfittare quando si utilizza IISNode.

IISNode non fornisce supporto diretto per directory virtuali o applicazioni nidificate tramite la configurazione, per cui è necessario avvalersi di una funzionalità di IISNode che non fa parte della configurazione ed è molto meno conosciuta. Tutti i figli dell'elemento `<appSettings>` con `Web.config` vengono aggiunti all'oggetto `process.env` come proprietà utilizzando la chiave appSetting.

Consente di creare una directory virtuale nei nostri `<appSettings>`

```
<appSettings>
  <add key="virtualDirPath" value="/foo" />
</appSettings>
```

All'interno della nostra app Node.js possiamo accedere alle impostazioni `virtualDirPath`

```
console.log(process.env.virtualDirPath); // prints /foo
```

Ora che possiamo usare l'elemento `<appSettings>` per la configurazione, sfruttiamo questo e lo usiamo nel nostro codice server.

```
// Access the virtualDirPath appSettings and give it a default value of '/'
// in the event that it doesn't exist or isn't set
var virtualDirPath = process.env.virtualDirPath || '/';

// We also want to make sure that our virtualDirPath
// always starts with a forward slash
if (!virtualDirPath.startsWith('/', 0))
  virtualDirPath = '/' + virtualDirPath;

// Setup a route at the index of our app
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});
```

Possiamo usare virtualDirPath anche con le nostre risorse statiche

```
// Public Directory
server.use(express.static(path.join(virtualDirPath, 'public')));
// Bower
server.use('/bower_components', express.static(path.join(virtualDirPath,
'bower_components')));
```

Mettiamo tutto insieme

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

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

// Access the virtualDirPath appSettings and give it a default value of '/'
// in the event that it doesn't exist or isn't set
var virtualDirPath = process.env.virtualDirPath || '/';

// We also want to make sure that our virtualDirPath
// always starts with a forward slash
if (!virtualDirPath.startsWith('/', 0))
  virtualDirPath = '/' + virtualDirPath;

// Public Directory
server.use(express.static(path.join(virtualDirPath, 'public')));
// Bower
server.use('/bower_components', express.static(path.join(virtualDirPath,
'bower_components')));

// Setup a route at the index of our app
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});
```

Utilizzo di Socket.io con IISNode

Per ottenere Socket.io che funziona con IISNode, le uniche modifiche necessarie quando non si utilizza una directory virtuale / applicazione nidificata sono all'interno di `Web.config` .

Poiché Socket.io invia richieste che iniziano con `/socket.io` , IISNode deve comunicare a IIS che questi devono essere gestiti anche da IISNode e non sono solo richieste di file statici o altro traffico. Ciò richiede un `<handler>` diverso dalle app IISNode standard.

```
<handlers>
    <add name="iisnode-socketio" path="server.js" verb="*" modules="iisnode" />
</handlers>
```

Oltre alle modifiche apportate ai `<handlers>` è inoltre necessario aggiungere una regola di riscrittura URL aggiuntiva. La regola di riscrittura invia tutto il traffico `/socket.io` al nostro file server su cui è in esecuzione il server Socket.io.

```
<rule name="SocketIO" patternSyntax="ECMAScript">
    <match url="socket.io.+"/>
    <action type="Rewrite" url="server.js"/>
</rule>
```

Se si utilizza IIS 8, sarà necessario disabilitare l'impostazione webSockets in `Web.config` in aggiunta all'aggiunta delle regole di gestione e riscrittura di cui sopra. Questo non è necessario in IIS 7 poiché non esiste il supporto webSocket.

```
<webSocket enabled="false" />
```

Leggi Utilizzo di IISNode per ospitare le app Web Node.js in IIS online:
https://riptutorial.com/it/node-js/topic/6003/utilizzo-di-iisnode-per-ospitare-le-app-web-node-js-in-iis

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Node.js	4444, Abdelaziz Mokhnache, Abhishek Jain, Adam, Aeolingamenfel, Alessandro Trinca Tornidor, Aljoscha Meyer, Amila Sampath, Ankit Gomkale, Ankur Anand, arcs, Aule, B Thuy, baranskistad, Bundit J., Chandra Sekhar, Chezzwizz, Christopher Ronning, Community, Craig Ayre, David Gatti, Djizeus, Florian Hämmerle, Franck Dernoncourt, ganesshkumar, George Aidonidis, Harangue, hexacyanide, Iain Reid, Inanc Gumus, Jason, Jasper, Jeremy Banks, John Slegers, JohnnyCoder, Joshua Kleveter, KolesnichenkoDS, krishgopinath, Léo Martin, Majid, Marek Skiba, Matt Bush, Meinkraft, Michael Irigoyen, Mikhail, Milan Laslop, ndugger, Nick, olegzhermal, Peter Mortensen, RamenChef, Reborn, Rishikesh Chandra, Shabin Hashim, Shiven, Sibeesh Venu, sigfried, SteveLacy, Susanne Oberhauser, thefourtheye, theunexpected1, Tomás Cañibano, user2314737, Volodymyr Sichka, xam, zurfyx
2	Ambiente	Chris, Freddie Coleman, KlwntSingh, Louis Barranqueiro, Mikhail, sBanda
3	Analizzare gli argomenti della riga di comando	yrtimiD
4	API CRUD basata su REST semplice	Iceman
5	App Web con Express	Aikon Mogwai, Alex Logan, alexi2, Andres C. Viesca, Aph, Asaf Manassen, Batsu, bekce, brianmearns, Community, Craig Ayre, Daniel Verem, devnull69, Everettss, Florian Hämmerle, H. Pauwelyn, Inanc Gumus, jemiloii, Kid Binary, kunerd, Marek Skiba, Mikhail, Mohit Gangrade, Mukesh Sharma, Naeem Shaikh, Niklas, Nivesh, noob, Ojen, Pasha Rumkin, Paul, Rafal Wiliński, Shabin Hashim, SteveLacy, tandrewnichols, Taylor Ackley, themole, tverdohleb, Vsevolod Goloviznin, xims, Yerko Palma
6	Arresto grazioso	RamenChef, Sathish
7	Async / Await	Cami Rodriguez, Cody G., cyanbeam, Dave, David Xu, Dom Vinyard, m_callens, Manuel, nomanbinhussein, Toni Villena

8	async.js	David Knipe, devnull69, DrakaSAN, F. Kauder, jerry, lsampaio, Shriganesh Kolhe, Sky, walid
9	Autenticazione di Windows sotto node.js	CJ Harries
10	Autoreload su modifiche	ch4nd4n, Dean Rather, Jonas S, Joshua Kleveter, Nivesh, Sanketh Katta, zurfyx
11	Biblioteca Mongoose	Alex Logan, manuerumx, Mikhail, Naeem Shaikh, Qiong Wu, Simplans, Will
12	Buon stile di codifica	Ajitej Kaushik, RamenChef
13	CLI	Ze Rubeus
14	Codice Node.js per STDIN e STDOUT senza utilizzare alcuna libreria	Syam Pradeep
15	Come vengono caricati i moduli	RamenChef, umesh
16	Comunicazione Arduino con nodeJs	sBanda
17	Comunicazione client-server	Zoltán Schmidt
18	Comunicazione Socket.io	Forivin, N.J.Dawson
19	Connetti a Mongodb	FabianCook, Nainesh Raval, Shriganesh Kolhe
20	Consegna HTML o qualsiasi altro tipo di file	Himani Agrawal, RamenChef, user2314737
21	Creazione di API con Node.js	Mukesh Sharma
22	Creazione di una libreria Node.js che supporti entrambe le promesse e le callback first-error	Dave
23	Database (MongoDB	zurfyx

	con Mongoose)	
24	Debug dell'applicazione Node.js	4444, Alister Norris, Ankur Anand, H. Pauwelyn, Matthew Shanley
25	Debug remoto in Node.JS	Rick, VooVoo
26	Design API restful: best practice	fresh5447, nilakantha singh deo
27	Disinstallazione di Node.js	John Vincent Jardin, RamenChef, snuggles08, Trevor Clarke
28	Distribuzione dell'applicazione Node.js senza tempi di inattività.	gentlejo
29	Distribuzione di applicazioni Node.js in produzione	Apidcloud, Brett Jackson, Community, Cristian Boariu, duncanhall, Florian Hämmerle, guleria, haykam, KlwntSingh, Mad Scientist, MatthieuLemoine, Mukesh Sharma, raghu, sjmarshy, tverdohleb, tyehia
30	ECMAScript 2015 (ES6) con Node.js	David Xu, Florian Hämmerle, Osama Bari
31	Emettitori di eventi	DrakaSAN, Duly Kinsky, Florian Hämmerle, jamescostian, MindlessRanger, Mothman
32	Esecuzione di file o comandi con Child Processes	guleria, hexacyanide, iSkore
33	Esecuzione di node.js come servizio	Buzut
34	Esportazione e consumo di moduli	Aminadav, Craig Ayre, cyanbeam, devnull69, DrakaSAN, Fenton, Florian Hämmerle, hexacyanide, Jason, jdrydn, Loufylouf, Louis Barranqueiro, m02ph3u5, Marek Skiba, MrWhiteNerdy, MSB, Pedro Otero, Shabin Hashim, tkone, uzaif
35	Esportazione e importazione del modulo in node.js	AndrewLeonardi, Bharat, commonSenseCode, James Billingham, Oliver, sharif.io, Shog9
36	Eventloop	Kelum Senanayake

37	Evita l'inferno del callback	tyehia
38	File system I / O	4444, Accepted Answer, Aeolingamenfel, Christophe Marois, Craig Ayre, DrakaSAN, Duly Kinsky, Florian Hämmerle, gnerkus, Harshal Bhamare, hexacyanide, jakerella, Julien CROUZET, Louis Barranqueiro, midnightsyntax, Mikhail, peteb, Shiven, still_learning, Tim Jones, Tropic, Vsevolod Goloviznin, Zanon
39	Gestione degli errori di Node.js	Karlen
40	Gestire la richiesta POST in Node.js	Manas Jayanth
41	Gestore pacchetti filati	Andrew Brooke, skiilaa
42	grugnito	Naeem Shaikh, Waterscroll
43	Guida per principianti NodeJS	Niroshan Ranapathi
44	Hack	signal
45	http	Ahmed Metwally
46	Iniezione di dipendenza	Niroshan Ranapathi
47	Iniziare con la profilatura dei nodi	damitj07
48	Installare Node.js	Alister Norris, Aminadav, Anh Cao, asherbar, Batsu, Buzut, Chance Snow, Chezzwizz, Dmitriy Borisov, Florian Hämmerle, GilZ, guleria, hexacyanide, HungryCoder, Inanc Gumus, Jacek Labuda, John Vincent Jardin, Josh, KahWee Teng, Maciej Rostański, mmhyamin, Naing Lin Aung, NuSkooler, Shabin Hashim, Siddharth Srivastva, Sveratum, tandrewnichols, user2314737, user6939352, V1P3R, victorkohl
49	Instradare richieste ajax con Express.JS	RamenChef, SynapseTech
50	Integrazione con Mongodb	cyanbeam, FabianCook, midnightsyntax
51	Integrazione con MySQL	Aminadav, Andrés Encarnación, Florian Hämmerle, Ivan Schwarz, jdrydn, JohnnyCoder, Kapil Vats, KlwntSingh, Marek

		Skiba, Rafael Gadotti Bachovas, RamenChef, Simplans, Sorangwala Abbasali, surjikal
52	Integrazione del passaporto	Ankit Rana, Community, Léo Martin, M. A. Cordeiro, Rupali Pemare, shikhar bansal
53	Integrazione di Cassandra	Vsevolod Goloviznin
54	Integrazione di PostgreSQL	Niroshan Ranapathi
55	Integrazione MongoDB per Node.js / Express.js	William Carron
56	Interagire con la console	ScientiaEtVeritas
57	Invia notifica Web	Houssem Yahiaoui
58	Invio di un flusso di file al client	Beshoy Hanna
59	Koa Framework v2	David Xu
60	La gestione delle eccezioni	KlwntSingh, Nivesh, riyadhalnur, sBanda, sjmarshy, topheman
61	Le notifiche push	Mario Rozic
62	Linea di lettura	4444, Craig Ayre, Florian Hämmerle, peteb
63	Localizzazione del nodo JS	Osama Bari
64	Lodash	M1kstur
65	Loopback - Connettore basato REST	Roopesh
66	Mantenere costantemente attiva un'applicazione di nodo	Alex Logan, Bearington, cyanbeam, Himani Agrawal, Mikhail, mscdex, optimus, pietrovismara, RamenChef, Sameer Srivastava, somebody, Taylor Swanson
67	metalsmith	RamenChef, vsjn3290ckjnaoij2jikndckjb
68	Modulo Cluster	Benjamin, Florian Hämmerle, Kid Binary, MayorMonty, Mukesh

		Sharma, riyadhalnur, Vsevolod Goloviznin
69	MSSQL Intergration	damitj07
70	multithreading	arcs
71	N-API	Parham Alvani
72	Node server senza framework	Hasan A Yousef, Taylor Ackley
73	Node.js (express.js) con il codice di esempio angular.js	sigfried
74	Node.js Architecture & Inner Workings	Ivan Hristov
75	Node.js con CORS	Buzut
76	Node.JS con ES6	Inanc Gumus, xam, ymz, zurfyx
77	Node.js con Oracle	oliolioli
78	Node.js Design Fundamental	Ankur Anand, pietrovismara
79	Node.JS e MongoDB.	midnightsyntax, RamenChef, Satyam S
80	Node.js v6 Nuove funzionalità e miglioramenti	creyD, DominicValenciana, KlwntSingh
81	NodeJS con Redis	evalsocket
82	NodeJS Frameworks	dthree
83	npm	Abhishek Jain, AJS, Amreesh Tyagi, Ankur Anand, Asaf Manassen, Ates Goral, ccnokes, CD.., Cristian Cavalli, David G., DrakaSAN, Eric Fortin, Everettss, Explosion Pills, Florian Hämmerle, George Bailey, hexacyanide, HungryCoder, Ionică Bizău, James Taylor, João Andrade, John Slegers, Jojodmo, Josh, Kid Binary, Loufylouf, m02ph3u5, Matt, Matthew Harwood , Mehdi El Fadil, Mikhail, Mindsers, Nick, notgiorgi, num8er, oscarm, Pete TNT, Philipp Flenker, Pieter Herroelen, Pyloid, QoP, Quill, Rafal Wiliński, RamenChef, Ratan Kumar, RationalDev, rdegges, refaelos, Rizowski, Shiven, Skanda, Sorangwala Abbasali, still_learning, subbu, the12, tlo, Un3qual, uzaif, VladNeacsu, Vsevolod Goloviznin, Wasabi Fan, Yerko

		Palma
84	nvm - Node Version Manager	cyanbeam, guleria, John Vincent Jardin, Luis González, pranspach, Shog9, Tushar Gupta
85	OAuth 2.0	tyehia
86	package.json	Ankur Anand, Asaf Manassen, Chance Snow, efeder, Eric Smekens, Florian Hämmerle, Jaylem Chaudhari, Kornel, lauriys , mezzode, OzW, RamenChef, Robbie, Shabin Hashim, Simplans, SteveLacy, Sven 31415, Tomás Cañibano, user6939352, V1P3R, victorkohl
87	parser csv nel nodo js	aisflat439
88	passport.js	Red
89	Pool di connessione Mysql	KlwntSingh
90	Prestazioni Node.js	Florian Hämmerle, Inanc Gumus
91	Programmazione asincrona	Ala Eddine JEBALI, cyanbeam, Florian Hämmerle, H. Pauwelyn, John, Marek Skiba, Native Coder, omgimanerd, slowdeath007
92	Programmazione sincrona contro asincrona in nodejs	Craig Ayre, Veger
93	Promesse Bluebird	David Xu
94	Protezione delle applicazioni Node.js	akinjide, devnull69, Florian Hämmerle, John Slegers, Mukesh Sharma, Pauly Garcia, Peter G, pranspach, RamenChef, Simplans
95	Quadri di modelli	Aikon Mogwai
96	Quadri di test unitari	David Xu, Florian Hämmerle, skiilaa
97	Richiamata per promettere	Clement JACOB, Michael Buen, Sanketh Katta
98	Richiedere()	Philip Cornelius Glover
99	Route-Controller-Struttura del servizio per ExpressJS	nomanbinhussein

100	Routing NodeJs	parlad neupane
101	Sequelize.js	Fikra, Niroshan Ranapathi, xam
102	Sfide di prestazione	Antenka, SteveLacy
103	Socket TCP	B Thuy
104	Storia di Nodejs	Kelum Senanayake
105	Struttura del progetto	damitj07
106	Upload di file	Aikon Mogwai, Iceman, Mikhail, walid
107	Usa casi di Node.js	vintproykt
108	Usando i flussi	cyanbeam, Duly Kinsky, efeder, johni, KlwntSingh, Max, Ze Rubeus
109	Usare Browserfiy per risolvere l'errore 'richiesto' con i browser	Big Dude
110	Utilizzando WebSocket con Node.JS	Rowan Harley
111	Utilizzo di IISNode per ospitare le app Web Node.js in IIS	peteb