

Node.js, dalla teoria alla pratica

Proseguiamo l'esame di questa soluzione per il Web, open source, innovativa e molto funzionale.

Riteniamo che Node.js sia interessante perché è un server web costruito con un nucleo C++ ottimizzato e con la macchina virtuale V8, il motore JavaScript del browser Chrome. La scelta del linguaggio è motivata da diverse ragioni, di cui una tecnica: si tratta di un linguaggio funzionale (ci torneremo tra poco) in cui il flusso di controllo è basato su un unico thread. L'ambiente è open source ed è altrettanto amichevole sia nei confronti di Unix sia di Windows, quindi è molto facile trovarlo nelle piattaforme di hosting, in particolare Node.js non a caso è una delle opzioni nei servizi cloud di Microsoft e Amazon.

Ricordiamo che JavaScript è un linguaggio pienamente funzionale, fra l'altro perché una variabile può contenere una funzione esattamente come può contenere un numero. L'approccio funzionale è un requisito che nasce dalla filosofia di progetto di Node.js. Il motore di questo sistema supporta un solo flusso di controllo per tutte le connessioni e l'input e output non bloccante. La parallelizzazione delle attività per fare fronte a tutte le connessioni contemporaneamente attive è lasciata all'applicazione.

La filosofia di progetto è l'opposto di quella con cui è nato il server Apache,

sulle ceneri del server Ncsa, nel 1995 per opera di Brian Behlendorf e altri (http://apache.org/ABOUT_APACHE.html): un processo attivo per ogni connessione. Questa scelta, naturale per chi sviluppa su Unix è andata in crisi in pochi anni, per la crescita vertiginosa del web. Meno di cinque anni dopo, sul web girava la sfida per gestire diecimila connessioni simultanee (*c10k challenge*).

Per quanto possa sembrare contro intuitivo, uno approccio vincente per vincere la sfida delle connessioni è smantellare il meccanismo che appoggia sul kernel l'onere di gestire il parallelismo creando un processo per connessione. In questo schema il sistema operativo gestisce sia le connessioni di rete, sia il parallelismo dell'applicazione. Purtroppo, però, creare e eliminare processi in continuazione non è un'attività che il kernel può gestire in modo abbastanza efficiente. Ha prevalso, quindi, l'idea di lasciare gestire al sistema operativo lo scambio con la rete, cosa che il kernel fa in maniera spettacolarmente efficiente, e gestire il parallelismo dentro l'applicazione.

Node.js viene creato da Ryan Dahl nel 2009 intorno a un nucleo estremamente ottimizzato per gestire il colloquio con la rete in modo asincrono e non bloccante.

La prima sfida di Node.js

Analizziamo un po' più in dettaglio l'architettura di Node.js, rimandando chi volesse approfondire ulteriormente a un'eccellente compilation di risorse per partire con Node.js, raccolta sul sito per programmatori Stackoverflow,

all'indirizzo bit.ly/nodejs_intro.

La sfida che Node.js pone al programmatore inizia qui: c'è un solo flusso di controllo e nessuna attività deve essere bloccante, perché ogni operazione è in tempo reale e impedisce al server di ascoltare sulla rete.

Facciamo un esempio: se un utente esegue l'upload di un video su un sito come YouTube, la connessione può durare parecchio. Ovviamente il server deve consentire il caricamento di più video contemporaneamente e non può fermarsi fino a che un video è completato prima di aspettare il secondo. La soluzione è nella natura del colloquio in rete.

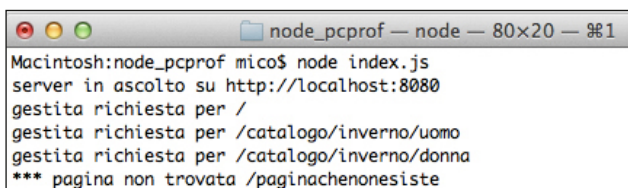
Se si osserva la connessione al livello appena sopra il sistema operativo, si vede un server che è in ascolto su un grande numero di socket, potenzialmente decine di migliaia, e legge dati da ogni socket quando questi sono disponibili.

L'invio di un file di grandi dimensioni, come il download della home page di un quotidiano, o l'upload di un video, avviene in realtà in diversi blocchi, le cui dimensioni sono dettate dal protocollo Tcp.

Il sistema operativo provvede alla lettura dei pacchetti, alla ricostruzione della sequenza dei dati inviati e all'allocazione di buffer e l'attribuzione di ogni pacchetto ricevuto a una determinata connessione, quindi risveglia il processo in attesa di dati e passa il buffer attivo. Ne segue che il colloquio in rete è sempre asincrono e avviene a pezzetti, anche se può capitare che un invio e una risposta siano atomici, se di piccole dimensioni.

Se un server vuole gestire in modo

La pagina iniziale della nostra applicazione.



Il log dell'applicazione, utile per le verifiche.

asincrono processi lunghi come l'upload di un video, deve quindi mantenere una lista delle connessioni aperte e dei relativi file in arrivo, quindi aspettare che arrivi un blocco di dati e scriverlo nel file giusto.

La seconda sfida di Node.js

Node.js è un'infrastruttura per creare dei server, non un framework su cui costruire.

Facciamo un esempio: se usiamo Php, creiamo un file *elencoProdotti.php* in una directory di nome *catalogo* della radice del server e daremo ai nostri clienti la visione di un *mio.server.it/catalogo/elencoProdotti.php*. Se il file esiste, Php ci risponderà a tono su questa richiesta, se il file manca avremo un errore 404. Occorre un certo lavoro per presentare Url come *mio.server.it/prodotti/cucina/apricatole_in_bronzo*.

Con Node.js è diverso, si crea un server eseguendo la funzione `createServer` e passando a questa una funzione per la gestione del dialogo

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

var server = function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
};

http.createServer(server).
listen(8080);
```

La funzione `createServer` crea l'infrastruttura che sta in ascolto sulla rete, mentre la funzione passata come parametro, che in questo caso si chiama proprio *server*, gestisce il dialogo. Nel nostro esempio, risponde con il consueto saluto a qualsiasi richiesta, per qualunque percorso, a partire da *mio.server.it:8080*.

La differenza con l'esempio Php è lampante: dobbiamo gestire esplicitamente la decodifica delle Url e l'instradamento del flusso di controllo in base al percorso richiesto sul sistema locale. Node.js non è l'unica infrastruttura server che si comporta così, quindi non si tratta né di un vantaggio esclusivo né di un difetto caratteristico, è solo un fatto. Si tratta di un po' più di lavoro, rispetto a quello che propongono altre architetture per il web, ma non poi così tanto. Questo lavoro extra è compensato con la libertà di organizzare il nostro sistema di percorsi come

meglio crediamo, con tutti gli alias che ci possono venire in mente e un eventuale schema Rest come vantaggio aggiunto, per esempio

Organizzare il lavoro

Senza farci intimidire, quindi, cominciamo a organizzarci.

Il codice che abbiamo mostrato poco sopra è quello da cui parte qualunque server

```
http.createServer(server).
listen(80);
```

la funzione che fa da parametro a `createServer` ha la responsabilità di provvedere a realizzare l'infrastruttura che risponde alle richieste in arrivo. Cosa ci occorre? Il primo ruolo che deve svolgere il nostro server è quello di inoltrare il routing della richiesta a un gestore specifico, basandosi sul percorso che la compone.

Il secondo ruolo, è quello di generare materialmente il contenuto e restituirlo al server.

Questo ha due conseguenze macroscopiche.

Da un lato, la funzione `server` ha la necessità di avere informazioni su uno schema di routing delle richieste, quello che si tradurrà nella struttura dei percorsi del nostro server.

In secondo luogo, dal fatto che la richiesta deve essere passata a una funzione di routing e da qui instradata alle funzioni che generano la risposta, discende che conviene passare per tutto il codice l'oggetto che descrive la richiesta, con i suoi parametri e i suoi cookie, facendogli attraversare diversi strati di responsabilità fino al risponditore effettivo.

Più avanti, dopo avere avuto la gioia di inventare la ruota, scopriremo che esiste un intero catalogo di ruote, di design e in lega leggera che funzionano perfettamente, come il framework *Express*. Prima, però, rendiamoci conto di cosa fa il framework per capire meglio.

Una tabella di percorsi

Cominciamo a scegliere una struttura dati adatta per la tabella di routing. Perché noi avremo una tabella e non penseremo nemmeno per scherzo di cavarcela con un nido di if in costante evoluzione. La struttura dati serve ad associare a un percorso, quindi a una stringa, una funzione, che produrrà il contenuto per il browser remoto.

La struttura dati ideale, sarebbe chiamata array associativo o map nella maggior parte dei linguaggi. In JavaScript, per qualche bizzarria della storia la struttura dati in effetti è un oggetto.

Proviamo, infatti, a creare dinamicamente un oggetto usando Node.js interattivamente

```
> var s = {};
undefined
> s.colore = "verde";
'verde'
> var f = function() { console.
log("ciao"); }
undefined
> s.o = f;
[Function]
> s
{ colore: 'verde', o: [Function] }
> s.o()
ciao
> s['colore']
'verde'
```

Vediamo con quale semplicità possiamo palleggiare funzioni e variabili in questo esempio e vediamo anche la libertà con cui possiamo alternare una sintassi stile Java (`s.colore`) e una stile array associativo Pythonico (`s['colore']`) per usare la stessa proprietà di un oggetto. Questo fa capire perché JavaScript è un linguaggio flessibile e adattabile, ideale anche per il server.

Quindi è assodato: se il nostro server deve avere una Url come *mio.server.it/catalogo/uomo/inverno*, dovremo scrivere nel codice qualcosa come

```
handler["catalogo/uomo/inverno"]
= requesthandlers.uomoinverno;
```

possiamo anche creare facilmente alias ogni volta che sia necessario, per esempio

```
handler["catalogo/inverno/
maschile"] = requesthandlers.
uomoinverno;
```

Ricapitoliamo: è facile creare una struttura dati a cui affidare l'instradamento delle richieste realizzando una tabella che associa percorsi e funzioni che restituiscono contenuto.

Esaminiamo un paio di vincoli che derivano da questa organizzazione.

Primo: la creazione della tabella deve essere posteriore alla definizione dei gestori di contenuti, altrimenti manca l'elemento da assegnare a un determinato percorso.

Secondo: la funzione che esegue il routing esamina la tabella e invoca il gestore appropriato per ogni contenuto richiesto, se ne abbiamo uno. Il gestore provvede autonomamente a inviare il contenuto al browser e, per farlo, ha bisogno di un riferimento ai dati della richiesta, come la connessione e i cookie, quindi ogni gestore di contenuto deve avere fra i parametri un oggetto request. Esaminiamo quindi come potrebbe essere il file iniziale del nostro server Node.js, chiamiamolo index.js

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");
```

```
var handler = {}
handler["/"] = requestHandlers.
home;
handler["/catalogo/inverno/
uomo"] = requestHandlers.
catalogo_inverno_uomo;
handler["/catalogo/inverno/
donna"] = requestHandlers.
catalogo_inverno_donna;
```

```
server.start(router.route,
handler);
```

Le tre istruzioni *require* all'inizio del file servono a importare le tre sezioni in cui abbiamo segmentato l'applicazione. Il primo file, *server*, provvede a gestire quello che è di pertinenza generale del server, dentro *router* sta la logica di instradamento delle richieste ai loro gestori e, infine, *requestHandlers* contiene le funzioni che provvedono a realizzare e inviare il contenuto.

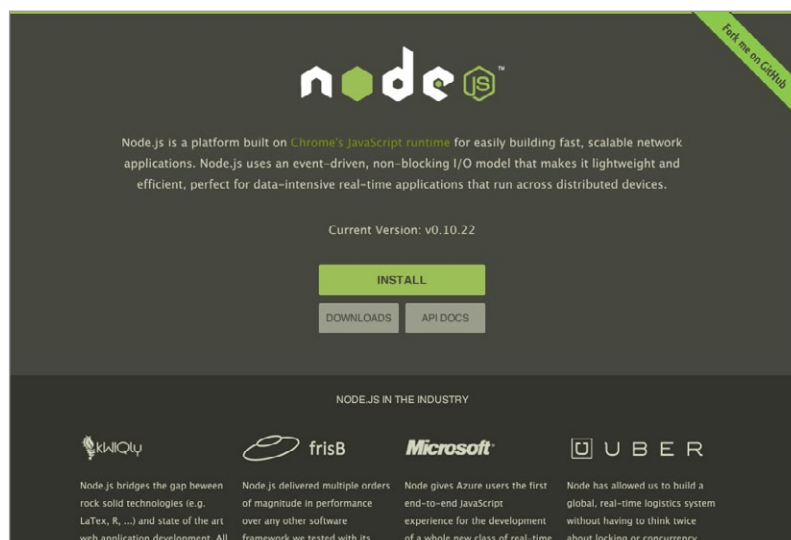
Questa separazione è solo una scelta di organizzazione e non è imposta dall'architettura del sistema o da un framework, visto anche che stiamo lavorando senza framework.

Subito dopo le *require* abbiamo la costruzione di una tabella di gestori di pagine, nell'oggetto *handler*, seguita dalla definizione della struttura dei nostri percorsi, che per ora sono solo tre, una radice e due cataloghi.

Infine, il server viene avviato passandogli la funzione e la tabella di routing. Vediamo ora la sequenza di avvio del server.

Il nucleo del server

Il codice completo del nostro server, nel file *server.js*, è piuttosto corto e lo riportiamo integralmente.



Questa è la pagina di riferimento per lo sviluppo nodejs.org

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

function start(route, handler) {
  function onRequest(request,
  response) {
    var pathname = url.
    parse(request.url).pathname;
    request.
    setEncoding("utf8");
    route(handler, pathname,
    response);
  }

  var port = 8080;
  http.createServer(onRequest).
  listen(port);
  console.log("server in
  ascolto su http://localhost:" +
  port);
}

exports.start = start;
```

Le prime due righe di *import* fanno riferimento a moduli di libreria di Node.js, che fanno parte del corredo standard: il gestore del protocollo e le funzioni di utilità per lavorare con le url. Segue la definizione della funzione *start*, con due parametri e la funzione di esportazione di questa funzione:

```
exports.start = start;
```

questo comporta che la funzione *start*, sarà esportata, sempre con il nome *start*. Il nome visibile all'esterno è quello a sinistra: nulla ci avrebbe impedito di scrivere *exports.inizio = start* per consentire l'uso della funzione *start* con il nome di inizio nei moduli che importano questo codice. Confrontiamo ora questo codice con quello che abbiamo visto nel paragrafo precedente:

```
var server = require("./server");
[...]
server.start(...);
```

Ed ecco come funziona il meccanismo della creazione di moduli: la variabile *server*, creata con l'istruzione *var server = ...* contiene come elementi *start*, perché il modulo esegue *exports.start = ...*. Semplice e efficace.

Cosa succede all'interno della funzione che inizializza il server? Innanzitutto, *start* ha due parametri: *route*, che è una funzione in grado di provvedere al routing di una determinata richiesta, *handler* è la tabella dei gestori dei percorsi che il server offrirà al mondo.

La funzione *route* è definita altrove e la vedremo in seguito, mentre la tabella *handler*, che, come abbiamo visto, è realizzata con le proprietà di un oggetto, usato come array associativo, è definita nell'inizializzazione dell'applicazione in *index.js*.

Esaminiamo il codice della funzione *start*, saltando per il momento la funzione *onRequest*.

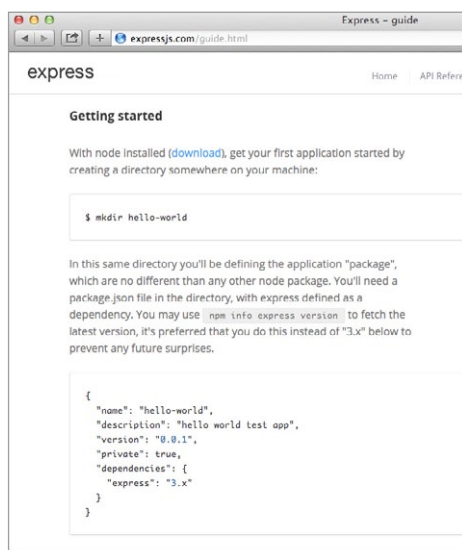
Sono tre righe di codice, che potrebbero essere condensate in una.

La variabile *port*, viene creata e inizializzata con un numero di porta, che serve a chi opera su sistemi operativi Unix, come per esempio OS X, in cui certi numeri di porta sono riservati agli utenti privilegiati. La porta 80 è una di queste porte riservate, mentre 8080 è per tradizione una porta di test.

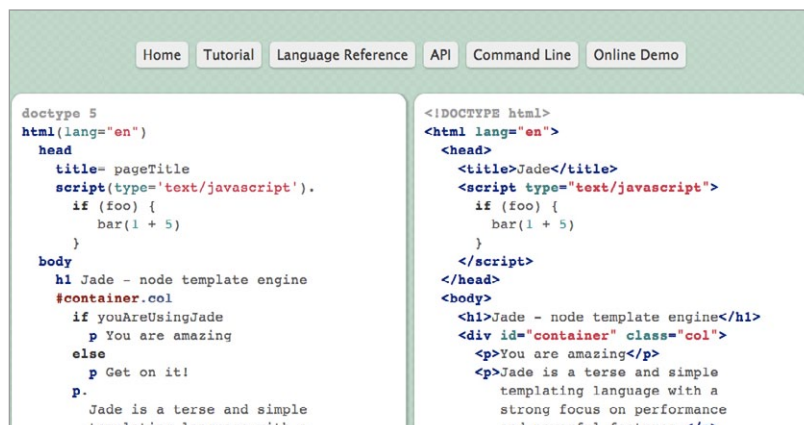
La creazione del server avviene in due step, uno in coda all'altro:

```
http.createServer(onRequest).
listen(port);
```

la creazione del server riceve come parametro una funzione, definita poco sopra. Il codice di creazione restituisce come valore il server appena creato, a cui si può



La home page del framework express per Node.js



L'eccellente motore di template Jade, indispensabile per creare pagine curate nello stile.

applicare il metodo `listen`, per scegliere una porta su cui stare in ascolto.

L'ultima riga è solo una stampa sul terminale, utile per il debugging primordiale. Passiamo adesso alla funzione passata al server per gestire le connessioni in arrivo, `onRequest`.

Si usa il metodo `url.parse` – ricordiamo la `require("url")` all'inizio del codice – per estrarre il `pathname` dalla url in arrivo, che è conservata nel campo `url` dell'oggetto `request`.

Il resto è semplice: si imposta un encoding per la richiesta, un'operazione da non dimenticare nella nostra lingua ricca di caratteri fuori dal set principale.

L'ultima riga, infine, invoca la funzione `route` passata fra i parametri, fornendo il path della uri richiesta, la tabella di gestori del contenuto e l'oggetto `response`, indispensabile ai content handler per restituire la risposta al browser collegato.

Il router

Vediamo il codice che provvede all'indirizzamento del contenuto verso i content handler.

Al livello più esterno si tratta solo della definizione di una funzione e della sua esportazione, secondo lo schema che abbiamo già visto.

La funzione di routing è molto semplice: un `if` discrimina fra due casi: quello in cui abbiamo un gestore per la pagina richiesta e quello in cui non lo abbiamo.

```
function route(handler, pathname, response) {
  if (typeof handler[pathname] === 'function') {
    console.log("gestita richiesta per " + pathname);
    return handler[pathname](response);
  } else {
    console.log("*** pagina
```

```
non trovata " + pathname);
    response.writeHead(404, {
      "Content-Type": "text/plain"
    });
    response.write("404 pagina non trovata");
    response.end();
  }
}
```

```
exports.route=route
```

Nel caso in cui manchi la funzione dedicata a servire la pagina richiesta, il codice risponde con una pagina di errore.

La funzione `writeHead` dell'oggetto `response` scrive un header di protocollo http corretto, con un codice di errore 404, seguito da un header che indica che il contenuto della pagina è puro testo. Il protocollo consente diversi header, quindi il secondo parametro di `writeHead` è un array di stringhe, inizializzato fra parentesi graffe.

Ricordiamo che gli header di protocollo non sono mostrati dal browser e non fanno parte della pagina, ma devono essere gestiti correttamente. Infatti, il browser mostrerà di aver ricevuto un errore solo se il protocollo indica errore, indipendentemente dal contenuto della pagina inviata.

Dopo l'intestazione di protocollo, una `response.write` invia al browser il contenuto della pagina in formato testo.

L'istruzione `response.end` provvede a chiudere la comunicazione e non lasciare in attesa il browser.

Se la uri richiesta è presente nella tabella di routing, il controllo viene trasferito alla funzione associata senza altri indugi.

Finalmente, i content handler

I nostri content handler, sono raccolti nel listato 1. Si tratta di codice molto semplice e ripetitivo.

Dato che la testata del blocco html restituito è la stessa per tutte le funzioni, l'abbiamo fattorizzata dichiarando esternamente una variabile che la contiene. Il corpo della funzione costruisce una stringa html e invoca il codice, uguale per tutti i casi gestiti, che invia un `write` uno header 200, per indicare successo, e quindi il testo html di risposta.

Questo codice è semplicistico per un caso reale, possiamo fare molto di più e molto meglio usando un modulo per la gestione di template Html, come l'ottimo Jade (<http://jade-lang.com>). Un template engine consente di creare pagine ricche, con le impostazioni di stile necessarie, fogli di stile ponderati e sostituzione dinamica di variabili di linguaggio all'interno della maschera predefinita.

Conclusioni

Abbiamo trattato solo la prima delle sfide che Node.js pone al programmatore: la responsabilità della strutturazione del codice e più cose da scrivere per organizzare il proprio codice. Per mostrare in dettaglio gli aspetti principali della formazione del contenuto e il flusso del controllo nel server, non abbiamo fatto riferimento a uno dei tanti framework per Node.js.

Un framework vero e proprio organizza il codice in modo più comprensivo e sofisticato di come abbiamo fatto noi, consentendo più completezza e flessibilità. In particolare, un template engine non è una soluzione di cui si può fare a meno. Ci rimane da coprire il secondo aspetto di Node.js di cui abbiamo accennato all'inizio, la necessità di organizzare il codice per callback evitando le situazioni in cui il server si può bloccare perché una funzione non ritorna immediatamente. Si tratta di un aspetto più avanzato, su cui ritorneremo. •