- Francesco Fullone - Enrico Zimuel -
- Federico Galassi - Matteo Collina -

PROGRAMMARECON

JavaScript

best practices

Testare il proprio codice tramite Mocha e Zombie.js >>

Migliorare le performance delle proprie applicazioni >>

Sfruttare Node.Js per creare applicazioni realtime >>

Utilizzare CoffeeScript come meta linguaggio>>

EDIZIONI FAG MILANO

*pro DigitalLifeStyle

PROGRAMMARECON

JavaScript

best practices

Francesco Fullone
Enrico Zimuel
Federico Galassi
Matteo Collina

EDIZIONI
FAG
MILANO

JavaScript | best practices

Gli autori: Francesco Fullone, Enrico Zimuel, Federico Galassi,Matteo Collina

Collana:

PROGRAMMARECON

Publisher: Fabrizio Comolli
Editor: Marco Aleotti
Progetto grafico e impaginazione: Roberta Venturieri
Coordinamento editoriale, prestampa e stampa: escom - Milano

ISBN: 978-88-6604-000-0

Finito di stampare in Italia presso Rotolito Lombarda - Seggiano di Pioltello (Mi) nel mese di 0000 2013

Sommario

- JAVASCRIPT E LA SUA COMMUNITY
- TESTARE JAVASCRIPT - ESEMPI E TECNOLOGIE
- PROGRAMMAZIONE ASINCRONA
- OTTIMIZZAZIONE DEL CODICE
- JAVASCRIPT DESIGN PATTERN
- STORAGE LOCALI
- NODE.JS
- COFFEESCRIPT

Introduzione

Con la continua evoluzione delle tecnologie legate al web è sempre più importante lavorare garantendo elevati standard di qualità, al fine di ridurre i costi di gestione delle applicazioni.
JavaScript Best Practices, il primo libro nel suo genere in italiano, affronta le più recenti tecnologie e le migliori pratiche di sviluppo con JavaScript dal punto di vista di alcuni riconosciuti professionisti del mondo web.

Chi dovrebbe leggere questo libro

Il libro è dedicato a tutti gli sviluppatori JavaScript, sia quelli più esperti sia quelli alle prime armi. *JavaScript Best Practices* è stato studiato come un ricettario da cui ognuno può tirare fuori la miglior ricetta e rielaborarla secondo le proprie necessità. Non solo teoria, ma anche molti esempi concreti e vere e proprie guide che permettono di mettere in pratica fin da subito quello che si è appreso.

Cosa contiene questo libro

Dopo un capitolo di introduzione sulla storia del linguaggio e la comunità degli sviluppatori, viene affrontato il primo dei due argomenti principali del libro, le buone pratiche di sviluppo, e in particolare il testing, la programmazione asincrona con JavaScript, l'ottimizzazione del codice e i pattern.
Il secondo argomento principale è costituito dalle librerie specifiche messe a disposizione dal linguaggio o che lo utilizzano per estenderne le potenzialità. Si approfondiscono CoffeeScript, Node.js e gli storage locali introdotti con HTML 5.

Cosa non contiene questo libro

Ci sarebbero tantissimi argomenti relativi a JavaScript, ma per motivi di spazio questo libro non riesce a trattarli tutti. Se il vostro interesse riguarda lo sviluppo front-end suggeriamo il libro **Programmare con JQuery** edito sempre da FAG.

Feedback e codice di esempio

I feedback sono importanti per capire in che misura siamo riusciti a raggiungere gli obiettivi prefissati e per fare ancora meglio la prossima volta.
Potete contattarci, e scaricare gli esempi contenuti nel libro, tramite il sito www.jsbestpractices.it e la relativa pagina Facebook www.facebook.com/jsbestpractices.

Risorse utili

La comunità di JavaScript è in continuo fermento ed è sempre possibile trovare interessanti risorse sul web.
Oltre al sito del GrUSP (www.grusp.org) e alla relativa mailing list vi consigliamo di tenere nei vostri bookmark dailyjs.com e html5rocks.com, dove sono pubblicati articoli e tutorial sempre aggiornati.
Non dimenticatevi che gran parte delle novità, e dei relativi approfondimenti arriva dalle conferenze come il JavaScript Day (jsday.it).

Ringraziamenti

Gli autori ringraziano il GrUSP per l'iniziativa e tutti coloro i quali, durante la stesura del libro, hanno dato il loro supporto morale e tecnico.

7

Node.js

di Matteo Collina

"Aerodynamically, the bumblebee should not be able to fly, but it does not know this, so it goes on."

Gabriele Lana, fondatore di Clean Code

Node.js (http://node.js.org) non dovrebbe esistere né essere efficace, poiché va contro tutti gli assunti della programmazione concorrente così come viene insegnata all'università. Eppure funziona.
L'autore, Ryan Dahl, racconta che Node.js nasce dall'ecosistema Ruby e in particolare dal lavoro di Zed Shaw su Mongrel, una libreria che consentiva a chiunque di avviare un veloce server web. Nato nel 2006, Mongrel ha rappresentato lo standard per il deployment di applicazioni Ruby on Rails, inoltre il suo codice è ancora alla base della maggior parte dei server web in ambiente Ruby.
Il problema di Ruby è che non è concorrente: la maggior parte dei server Ruby in quel periodo poteva eseguire solo una richiesta alla volta. La geniale intuizione di Ryan, che ha lasciato un Ph.D. per sviluppare Node.js, consiste nell'aver realizzato un server web totalmente asincrono, in grado di gestire molte connessioni contemporanee.
Nel 2009, l'anno in cui Ryan ha presentato Node.js alla JSConf.EU, JavaScript non era ancora stato portato sui server e mancavano tutte le primitive fondamentali come per esempio leggere un file da disco. Se fossero esistiti degli standard da seguire, secondo Ryan sarebbero stati redatti nel modo sbagliato e non gli avrebbero consentito di sviluppare Node.js.

L'esempio che ha reso famoso Node.js è un web server scritto in sole sei righe di codice:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

Node.js è stato realizzato su una lavagna bianca, partendo da V8 (la macchina virtuale JavaScript di Google Chrome) e aggiungendo la gestione concorrente che lo ha reso famoso. JavaScript è un linguaggio ideale per questo tipo di compiti poiché è nato con il concetto di evento, perfetto per implementare applicazioni asincrone e concorrenti. Node.js non è solo JavaScript server-side, ma è anche una piattaforma su cui eseguire 22.908 librerie (dati di febbraio 2013) pubblicate su NPM (http://npmjs.org). La forza della comunità Node.js è proprio questa: produrre tante piccole librerie che facciano una sola cosa estremamente bene. Questa è la pura filosofia UNIX, per cui difficilmente troverete il framework che fa qualsiasi cosa. Piuttosto Node.js vi darà tutti gli strumenti per creare applicazioni complesse.

Installazione

Installare Node.js è facile, che voi siate su Mac o su Windows. Aprite il vostro browser e inserite l'indirizzo http://nodejs.org. Cliccando su **Install** verrà scaricato il pacchetto di installazione per la vostra piattaforma. Eseguite l'istallazione e sarete pronti per iniziare. Su Windows per eseguire Node.js avrete bisogno di aprire un terminale: avviate l'eseguibile `cmd`, che potete trovare dal box di ricerca nel menu **Start**. Per avviare una shell Node.js basterà lanciare il comando `node`.
Se invece preferite la mela, eseguite una ricerca su Spotlight digitando "terminale" e avviate il Terminale. Per avviare una shell Node.js basterà lanciare il comando `node`.
Eseguendo il comando `node` avrete a disposizione un ambiente Node.js in cui provare snippet di codice, così:

```
$ node
> console.log("ciao mondo");
ciao mondo
undefined
>
```

Il comando `node` consente anche di eseguire un file JavaScript. Se infatti preparate un file `programma.js` contenente:

```
console.log("ciao mondo");
```

potete eseguirlo tramite `node programma.js` per ottenere l'output desiderato.

Se invece siete utenti Linux, vi consiglio di installare Node.js tramite NVM, che vedremo nel prossimo paragrafo, o mediante il package manager della vostra distribuzione, posto che la versione di Node.js sia almeno 0.8.x.

Node Version Manager

Node.js sta avendo un'evoluzione rapida, con molte release in un breve intervallo. Proprio per questo sono nate svariate soluzioni per spostarsi tra le varie versioni di Node.js con un semplice comando. Uno di questi strumenti è Node Version Manager (https://github.com/creationix/nvm), abbreviato NVM, disponibile per Mac e per Linux.
Gli utenti della mela morsicata devono installare Xcode e al suo interno i **Command Line Tools**, sotto al menu **Xcode**->**Preferences**->**Downloads**. Rilassatevi, sono più di 2 GB. Gli utenti Linux, invece, devono installare i pacchetti build-essential e libssl-dev. Dopo aver soddisfatto le dipendenze, installare NVM è semplicissimo:

```
$ curl https://raw.github.com/creationix/nvm/master/install.sh | sh
```

A questo punto possiamo installare l'ultima versione di Node.js disponibile (a febbraio 2013):

```
$ nvm install v0.8.20
```

Possiamo selezionare la versione di Node.js da utilizzare tramite un semplice:

```
$ nvm use v0.8.20
```

È consigliabile aggiornare la versione di default, in modo tale da non doverla selezionare a ogni avvio:

```
$ nvm alias default v0.8.20
```

Le basi di Node.js

Node.js consente lo sviluppo di applicazioni "lato server" in JavaScript sfruttando V8. Il supporto a JavaScript è perciò molto esteso, tipicamente al pari dell'ultima versione di Google Chrome rilasciata e, come vedremo nel seguito, include appieno EcmaScript 5. Questa specifica rappresenta l'ultima versione stabile del linguaggio di programmazione più popolare sul web. Node.js implementa la specifica common.js, che definisce il modo in cui vari "moduli" JavaScript interagiscono tra loro. Vedremo come questo influisce su Node.js e sul modo in cui costruiamo i nostri programmi. Node.js estende JavaScript aggiungendo tutta una serie di funzionalità specifiche per gestire le operazioni di I/O in modo totalmente asincrono. Vedremo come e perché, per fare questo, introduce una sua regola per la gestione degli errori.

Analizzeremo gli oggetti globali di Node.js e vedremo come evitare possibili problemi. Oltre a questo, gli altri aspetti che tratteremo in dettaglio saranno gli eventi e gli stream.

EcmaScript 5

Node.js supporta pienamente EcmaScript 5 e, tramite alcuni flag, anche alcune funzionalità del prossimo EcmaScript 6, nome in codice Harmony. La maggior parte dei lettori si chiederà cosa significa per loro, ma è importante notare che possiamo sfruttare tutte le potenzialità del linguaggio, in particolare gli iteratori e la gestione delle proprietà. Normalmente in JavaScript siamo abituati a scrivere cicli `for`, ma in Node.js possiamo scrivere:

```
[1, 2, 3, 4, 5].forEach (function (e) {
  console.log(e); // scrive sulla console l'elemento dell'array
});
```

Inoltre, abbiamo a disposizione anche i metodi `map` e `filter`:

```
[1, 2, 3, 4, 5].filter(function (e) {
  return e % 2 === 0; // scarta tutti i numeri dispari
}).map(function (e) {
  return e * 2;
}); // il risultato sarà [4, 8]
```

La gestione degli oggetti è molto più semplice. Per iterare sulle proprietà di un oggetto siamo abituati a scrivere:

```
var obj = { a: 2, b: 3 };
var p = null;

for (p in obj) {
  if (obj.hasOwnProperty(p)) {
    // do something intelligent with p
  }
}
```

Grazie al supporto a EcmaScript 5, in Node.js possiamo semplificare l'iterazione notevolmente senza preoccuparci di supportare versioni non compatibili dei browser:

```
var obj = { a: 2, b: 3 };
Object.keys(obj).forEach(function (p) {
  // do something intelligent with p
});
```

Common.js, require e i moduli

All'interno dei browser i file JavaScript vengono eseguiti tutti all'interno dello stesso scope, come se fossero concatenati. I browser non offrono alcun modo nativo per in-

cludere un file JavaScript dall'interno di un altro file JavaScript, limitando il suo scope. Questo modello non può essere sufficiente per lo sviluppo server side. Infatti, possiamo avere applicazioni molto più complesse e articolate che una "semplice" interazione con una pagina. Occorre perciò un metodo per consentire a più file JavaScript di interagire fra loro e di favorire il riuso.
Le applicazioni Node.js sono basate sulla specifica common.js, che definisce il formato e le modalità di caricamento di più "moduli". Un modulo altro non è che un file `.js` che durante la sua esecuzione esporta delle funzionalità agli altri moduli.
Anche se a prima vista può sembrare complicato, tramite un esempio possiamo analizzare meglio il concetto. Aprite il vostro editor preferito e preparate due file, `primo.js` e `incluso.js`.
All'interno di `incluso.js` scrivete:

```
module.exports.hello = function hello(name) {
  console.log("hello " + name);
};
hello("incluso.js");
```

L'oggetto `module.exports` sarà ciò che gli altri moduli vedranno.
All'interno di `primo.js` inserite quindi:

```
var hello = require("incluso").hello;
hello("Matteo");
```

Se lo eseguite, otterrete l'output voluto:

```
$ node primo.js
hello incluso.js
hello Matteo
```

Ora modificate il file `primo.js`:

```
var hello = require("incluso").hello;
hello("Matteo");
console.log(hello === require("incluso").hello);
```

Eseguendolo, ottenete:

```
$ node primo.js
hello incluso.js
hello Matteo
true
```

La funzione `require` richiede i moduli solo una volta e non li riesegue alle successive richieste, ma memorizza solo quanto esportato.

Gestione degli errori

Gestire gli errori nel modo corretto è la ricetta più efficace per un'applicazione Node.js di successo. Purtroppo un solo errore non gestito correttamente può causare il crash dell'intero processo Node.js, troncando le centinaia di richieste concorrenti che stava servendo in quel momento. Per questo motivo in Node.js non vengono tipicamente usate le eccezioni, lo strumento principe di tutti i linguaggi di programmazione.
Node.js è differente, proprio perché è asincrono. Consideriamo un compito semplice, come per esempio la lettura di un file. La condizione di errore più comune per questo task è la mancanza del file su disco: abbiamo specificato un path che non esiste.
In un linguaggio sincrono aprire un file inesistente causa un'eccezione immediata. In un linguaggio asincrono come Node.js, aprire un file inesistente non può lanciare un'eccezione. Poiché la lettura è asincrona, anche la condizione di errore lo sarà. Per esempio, la lettura di un file in Node.js si svolge nel modo seguente:

```
// carica il modulo file, per interagire con il filesystem
var fs = require("fs");
var file = "non-existent-path";

fs.readFile(file, function (err, data) {
  if (err) {
    // fai qualcosa per gestire l'errore
    console.log("file non trovato");
    return;
  }
  console.log(data.toString());
});

console.log("apro il file " + file);
```

Se proviamo a eseguire questo codice, vedrete che stamperà:

```
apro il file non-existent-path
file non trovato
```

Da questo esempio vediamo come l'esecuzione della callback viene eseguita in modo asincrono, ovvero dopo che il file ha completato la sua esecuzione. Inoltre, la callback viene invocata con due parametri: `err` e `data`. Il primo è di classe `Error`, mentre il secondo è di tipo `Buffer`:

- `Error` rappresenta il contesto dell'errore, infatti possiamo sempre lanciare l'eccezione `throw new Error("qualcosa è andato storto");`
- `Buffer` rappresenta un modo molto efficace di gestire dati binari in Node.js e a esso di deve una parte delle performance prodigiose di questo framework.

Oggetti globali

Ogni modulo Node.js è eseguito in una propria funzione, perciò le variabili definite tramite `var` sono a uso esclusivo del modulo in esame. Al contrario, le variabili senza `var` saranno globali e visibili da tutti i moduli Node.js. Per tracciarle meglio, Node.js offre un oggetto `global`, cui tutte le variabili globali vengono "attaccate". Node.js fornisce già alcune variabili globali, per esempio:

- `process`, un oggetto che rappresenta il processo corrente;
- `Buffer`, la classe con cui gestire dati binari in Node.js;
- `require`, per caricare nuovi moduli;
- `setTimeout`, `clearTimeout`, `setInterval` e `clearInterval`, per eseguire funzioni a intervalli di tempo prefissati;
- `console`, per stampare a video.

Possiamo vedere come funzionano le variabili globali tramite il seguente esempio. Scriviamo nel file `a.js`:

```
a = "hello world";
console.log(a);
console.log(global.a);
```

e nel file `b.js`:

```
console.log(global.a);

require("./a");

console.log(a);
console.log(global.a);
```

Lanciando node `b.js` otterremo l'output:

```
b1 undefined
a1 hello world
a2 hello world
b2 hello world
b3 hello world
```

Nell'esempio precedente è possibile vedere come la variabile globale `a`, dopo essere stata definita in un file, venga "passata" agli altri.
Infine, possiamo riscrivere il nostro esempio per prendere il file da leggere dalla riga di comando:

```
// carica il modulo file, per interagire con il filesystem
var fs = require("fs");
```

```
// l'ultimo argomento passato via riga di comando
// è il file da leggere
var file = process.argv[process.argv.length - 1];

fs.readFile(file, function (err, data) {
  if (err) {
    // terminiamo l'esecuzione del nostro programma
    // node con una condizione di errore != 0
    process.exit(1);
  }

  // possiamo usare lo standard output, che è
  // un canale binario e accetta un Buffer
  // come input
  process.stdout.write(data);
});
```

Tra le funzionalità fondamentali di Node.js vi è la possibilità di ritardare l'esecuzione di una funzione al prossimo "tick". Node.js opera all'interno di un event-loop, in altre parole un ciclo infinito che esegue una determinata funzione al verificarsi di un evento. Un "tick" è semplicemente un "giro" all'interno di questo ciclo. Possiamo avvalerci di questa funzionalità tramite l'uso di `process.nextTick`, come nel seguente esempio:

```
process.nextTick(function () {
  console.log("second");
});
console.log("first");
```

Node.js consente anche di usare le classiche funzioni JavaScript `setTimeout` e `setInterval` e le corrispettive per la cancellazione, `clearTimeout` e `clearInterval`.

EventEmitter e Streams

La vera novità di Node.js sta nell'aver implementato in JavaScript nuove API per la gestione degli eventi e per i flussi di dati. La classe `EventEmitter` è implementata dentro il modulo `events` e rappresenta la base della maggior parte dei moduli Node.js. Per esempio:

```
var EventEmitter = require("events").EventEmitter;

var e = new EventEmitter();

e.on("mio evento", function () {
  console.log("mio evento ricevuto");
});

e.emit("mio evento");
e.emit("mio evento");

process.exit(0);
```

Possiamo ascoltare un evento, identificato da una semplice stringa, semplicemente chiamando il metodo `on` e passando una funzione, che sarà chiamata al verificarsi dell'evento. Eseguendo questo esempio possiamo notare che l'esecuzione delle funzioni registrate è immediata.
Similmente a on, il metodo once consente di registrare una funzione che sarà chiamata al verificarsi di un evento, ma al più una sola volta. Per esempio:

```
var EventEmitter = require("events").EventEmitter;

var e = new EventEmitter();

e.once("mio evento", function () {
  console.log("mio evento ricevuto");
});

e.emit("mio evento");
e.emit("mio evento");

process.exit(0);
```

Basati sugli eventi, gli stream di Node.js consentono di ricevere i dati a mano a mano che vengono letti. Gli stream sono infatti alla base della gestione della concorrenza di Node.js. In particolare, mediante uno stream possiamo registrarci a più eventi:

- `data` viene emesso quando è stato letto dallo stream un nuovo pacchetto di dati;
- `end` viene emesso quando lo stream si chiude;
- `error` viene emesso se è rilevato un errore nello stream.

Possiamo quindi riscrivere il nostro esempio di lettura da file usando gli stream:

```
// carica il modulo file, per interagire con il filesystem
var fs = require("fs");

// l'ultimo argomento passato via riga di comando
// è il file da leggere
var file = process.argv[process.argv.length - 1];

var stream = fs.createReadStream(file);

stream.on("error", function () {
  // terminiamo l'esecuzione del nostro programma
  // node con una condizione di errore != 0
  process.exit(1);
});

stream.on("data", function (data) {
  // possiamo usare lo standard output, che è
  // un canale binario e accetta un Buffer
  // come input
  process.stdout.write(data);
```

```
});

stream.on("end", function (data) {
  process.exit(0);
});
```

Possiamo semplificare ulteriormente il nostro codice usando il metodo `pipe`:

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

// l'ultimo argomento passato via riga di comando
// è il file da leggere
var file = process.argv[process.argv.length - 1];

var stream = fs.createReadStream(file);

stream.pipe(process.stdout);

stream.on("error", function () {
  // terminiamo l'esecuzione del nostro programma
  // node con una condizione di errore != 0
  process.exit(1);
});
```

Il metodo `pipe` gestisce per noi la copia tra uno stream in lettura (o `ReadStream`) e uno in scrittura (o `WriteStream`).

Le librerie necessarie

La vera forza di Node.js è la comunità: il Node Package Manager fornisce più di 22.900 pacchetti (dati di febbraio 2013), che consentono di adattare JavaScript agli usi più disparati. Infatti, come già accennato, la comunità sposa la filosofia UNIX: realizzare e distribuire piccoli componenti riutilizzabili. Spesso i pacchetti pubblicati sono composti da un solo modulo.
Installare un pacchetto è facile. Basta lanciare il comando:

```
$ npm install <PACCHETTO>
```

È importante notare che il pacchetto verrà installato nella directory corrente, sotto la cartella `node_modules/<PACCHETTO>`. Se vogliamo installarlo globalmente dobbiamo indicarlo esplicitamente:

```
$ npm install <PACCHETTO> -g
```

Installare i pacchetti globalmente è una "bad practice". L'unica eccezione sono alcuni strumenti che sono espressamente studiati per essere installati in questo modo e che emettono un avviso se installati diversamente. Fra questi troviamo Mocha, Grunt e Bower.

I pacchetti possono includere script eseguibili. Nel caso di un'installazione "locale", li troveremo nella cartella `node_modules/.bin/`. Al contrario, nel caso di un'installazione globale, saranno inclusi direttamente nel PATH.
In questa parte del capitolo parliamo delle librerie più utilizzate e popolari per Node.js, quelle che si trovano nella "top ten" di utilizzo. Non discuteremo di tutte le loro funzionalità, ma piuttosto tratteremo quelle che si ritengono le più comuni.

Async

Async (http://github.com/caolan/async) è sicuramente una delle librerie più utili di tutto l'ecosistema Node.js. Se si adotta lo stile delle callback di Node.js, così come descritto nei paragrafi precedenti, non è possibile farne a meno. Per installarla, basterà digitare:

```
$ npm install async
```

Immaginiamo di voler estendere il nostro comando `cat` per concatenare sullo standard output tutti i file che gli verranno passati dalla riga di comando. Se fossimo in linguaggio tradizionale, itereremmo sugli argomenti e leggeremmo i file a uno a uno.
Per leggere tre file in sequenza possiamo eseguire in Node.js:

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

fs.readFile("a", function (err, dataA) {
  fs.readFile("b", function (err, dataB) {
    fs.readFile("c", function (err, dataC) {
      // esegui qualcosa con il risultato dei tre file.
    });
  });
});
```

Come possiamo vedere nell'esempio, le callback sono spesso innestate, creando un effetto "ad albero" che riduce la leggibilità. Siccome Node.js è una piattaforma asincrona, visto il suo successo, deve esistere un modo migliore per garantire la leggibilità in questo tipo di situazioni.
Fortunatamente Async ci viene in aiuto e, infatti, possiamo usare la funzione `async.series` per eseguire un array di funzioni asincrone in serie:

```
var fs = require("fs");
// caricare i pacchetti è equivalente a caricare un
// modulo
var async = require("async");

// i primi due argomenti sono il nome dell'eseguibile "node"
// e il nome dello script che stiamo eseguendo
var files = process.argv.splice(2);
```

```
async.series(files.map(function (file) {
  // per ogni file creiamo una nuova funzione anonima,
  // partendo da fs.readFile e aggiungendo il parametro
  // file.
  return async.apply(fs.readFile, file);
}, function (err, results) {
  if (err) {
    console.log(err);
  }
  // results è un array di buffer,
  // con il contenuto di tutti i file in ordine
  console.log(results.join(""));
});
```

Analizzando questo esempio è possibile notare che la funzione `async.series`, come la maggior parte delle funzioni di Async, ha due parametri: un array e una funzione. L'array contiene le funzioni che svolgeranno il lavoro, mentre il secondo parametro sarà invocato al termine di tutte le funzioni contenute nell'array.
Nota bene: tutte queste funzioni seguono la definizione di callback di Node.js, con l'errore come primo parametro. Nel caso di `async.series` i parametri successivi saranno passati alla funzione di notifica come array.
Oltre a `async.series`, Async offre un'ampia gamma di funzioni per svolgere in modo asincrono la maggior parte delle operazioni che in altri linguaggi avrebbero richiesto un ciclo. Per esempio `async.parallel` è simile in tutto e per tutto a `async.series`, ma esegue le funzioni asincrone in modo concorrente, per poi chiamare la funzione di notifica una volta che tutte le funzioni nell'array abbiano completato la loro esecuzione.

Underscore

Underscore è una libreria molto popolare tra gli sviluppatori front-end. Aggiunge, infatti, a JavaScript tante funzioni di utilità che non sono presenti nel linguaggio. Installarla, anche in questo caso, è semplicissimo:

```
$ npm install underscore
```

Per esempio, per recuperare il primo elemento di un array possiamo eseguire:

```
var _ = require("underscore");
console.log(_.first(["a","b","c"]));
```

Uno degli esempi più interessanti viene direttamente dalla documentazione di Underscore:

```
var _ = require("underscore");

var lyrics = [
  {line : 1, words : "I'm a lumberjack and I'm okay"},
```

```
  {line : 2, words : "I sleep all night and I work all day"},
  {line : 3, words : "He's a lumberjack and he's okay"},
  {line : 4, words : "He sleeps all night and he works all day"}
];

_.chain(lyrics)
  .map(function(line) { return line.words.split(' '); })
  .flatten()
  .reduce(function(counts, word) {
    counts[word] = (counts[word] || 0) + 1;
    return counts;
  }, {})
  .value();

=> {lumberjack : 2, all : 4, night : 2 ... }
```

In questo esempio l'oggetto `lyrics` viene "incapsulato" tramite la chiamata a `chain` aumentandolo dei metodi forniti da Underscore. Ogni `line` è poi trasformata tramite la funzione `map` in modo che ritorni un array contenente tutte le parole incluse nella riga. Mediante `flatten`, l'array di array così composto è trasformato in un unico array, contenente tutti i valori. Infine, come abbiamo già visto, la funzione `reduce` può essere usata per compiere il conteggio di ogni parola trovata.

Request

Request è la più semplice libreria per compiere chiamate HTTP a servizi REST. Inoltre, Request contiene tante funzionalità utili nel momento in cui si vogliono integrare servizi di terze parti in un'applicazione Node.js.
Per installare Request:

```
$ npm install request
```

Recuperare la pagina di ricerca di Google non è facile:

```
var request = require('request');
request('http://www.google.com', function (error, response, body) {
  if (!error && response.statusCode == 200) {
    console.log(body) // Print the google web page.
  }
});
```

Request ci permette inoltre di utilizzare qualsiasi verbo HTTP. Per esempio, possiamo fare una POST tramite:

```
request.post('http://service.com/upload').form({key:'value'});
```

Infine supporta anche l'autenticazione OAuth, per esempio per chiamare le API di Twitter o Facebook.

Commander

Commander è probabilmente la migliore libreria Node.js per la lettura dei parametri da riga di comando. Fornisce, infatti, un semplice approccio per leggere gli argomenti, specificare valori di default e infine generare un menu di aiuto. Come tutti i pacchetti pubblicati su NPM, installare questa libreria richiede un singolo comando:

```
$ npm install commander
```

Tornando all'esempio sui file, possiamo usare Commander per rendere il nostro comando semplice da utilizzare:

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

program
  .version('0.0.1')
  .usage('[options] <file ...>')
  .option('-n, --newline', 'print a newline at the end')
  .parse(process.argv);

var files = program.args;

var next = function () {
  var file = files.shift();
  if (!file) {
    if (program.newline) {
      console.log("");
    }
    process.exit(0);
  }

  var stream = fs.createReadStream(file);

  // copia tutto lo stream, ma non chiudere
  // la destinazione
  stream.pipe(process.stdout, { end: false });

  // in caso di errore
  stream.on("error", function () {
    // terminiamo l'esecuzione del nostro programma
    // node con una condizione di errore != 0
    process.exit(1);
  });

  // quando lo stream finisce
  // passiamo al file successivo
  stream.on("end", next);
};

next();
```

Inoltre, Commander consente di leggere tutti i possibili tipi di opzioni che possono essere utili al programma:

```
var program = require("commander");

function range(val) {
  return val.split('..').map(Number);
}

function list(val) {
  return val.split(',');
}

program
  .version('0.0.1')
  .usage('[options] <file ...>')
  .option('-i, --integer <n>', 'An integer argument', parseInt)
  .option('-f, --float <n>', 'A float argument', parseFloat)
  .option('-r, --range <a>..<b>', 'A range', range)
  .option('-l, --list <items>', 'A list', list)
  .option('-o, --optional [value]', 'An optional value')
  .parse(process.argv);

console.log(' int: %j', program.integer);
console.log(' float: %j', program.float);
console.log(' optional: %j', program.optional);
program.range = program.range || [];
console.log(' range: %j..%j', program.range[0], program.range[1]);
console.log(' list: %j', program.list);
console.log(' args: %j', program.args);
```

Realizzare una chat in tempo reale

Node.js può essere usato per implementare qualsiasi tipo di programma, ma la sua specialità sono le applicazioni server-side ad alto numero di connessioni in contemporanea.

Il classico esempio di questo tipo di progetti è la chat, che ci permetterà di vedere all'opera alcune delle molteplici possibilità di Node.js in maniera semplice e immediata. Vedremo anche alcune librerie client-side, in modo da realizzare un prototipo funzionante. L'esempio completo è disponibile all'indirizzo https://github.com/mcollina/js-best-practices-real-time-chat.

La trattazione partirà dall'analisi del file `package.json`, la struttura fondamentale di ogni applicazione Node.js, per poi affrontare il framework web Express e vedere come integrare Bootstrap in un'applicazione Node.js, usando Bower e Grunt. Infine per la comunicazione tra il browser e il server sarà usato Socket.io.

Il file package.json

Il file `package.json` rappresenta il meccanismo di definizione di ogni applicazione Node.js. In particolare consente di definire:

- il nome dell'app, gli autori, i riferimenti web e tutti gli altri possibili metadati, spesso utili se stiamo sviluppando una libreria;
- un flag `private`, per evitare che l'app venga pubblicata accidentalmente su NPM;
- tutte le dipendenze della nostra applicazione o libreria, in modo che siano automaticamente installabili. In particolare sono specificati sia il nome del pacchetto sia le versioni;
- lo script principale della libreria, che sarà ritornato dalla chiamata a `require` del pacchetto pubblicato su NPM;
- i comandi da invocare sia per avviare l'applicazione sia per eseguire i test.

Per creare un file `package.json` basta eseguire il seguente comando in una nuova directory:

```
$ mkdir js-best-practices-real-time-chat
$ npm init
```

Dopo aver risposto a una breve serie di domande, un file `package.json` verrà creato nella directory corrente:

```
{
  "name": "js-best-practices-real-time-chat",
  "version": "0.0.0",
  "description": "node.js example app for the Js Best Practices Book",
  "main": "index.js",
  "repository": {
    "type": "git",
    "url": "http://github.com/mcollina/js-best-practices-real-time-chat.git"
  },
  "keywords": [
    "example",
    "chat",
    "socket.io",
    "express",
    "bootstrap",
    "grunt",
    "bower"
  ],
  "author": "Matteo Collina <hello@matteocollina.com>",
  "license": "MIT",
  "readmeFilename": "README.md"
}
```

Aggiungiamo quindi uno script di avvio, così:

```
...
  "scripts": {
```

```
    "start": "node ./index.js"
  },
...
```

Eseguendo `npm start` otterremo l'errore:

```
> js-best-practices-real-time-chat@0.0.0 start
/Users/matteo/Repositories/js-best-practices-real-time-chat
> node ./index.js


module.js:340
    throw err;
          ^
Error: Cannot find module
'/Users/matteo/Repositories/js-best-practices-real-time-chat/index.js'
    at Function.Module._resolveFilename (module.js:338:15)
    at Function.Module._load (module.js:280:25)
    at Module.runMain (module.js:492:10)
    at process.startup.processNextTick.process._tickCallback
(node.js:244:9)
npm ERR! js-best-practices-real-time-chat@0.0.0 start: 'node ./index.js'
npm ERR! 'sh "-c" "node ./index.js"' failed with 1
npm ERR!
```

Aggiungiamo quindi un file `index.js` con il seguente contenuto:

```
console.log("Starting up");
```

E avremo così il seguente output:

```
$ npm start

> js-best-practices-real-time-chat@0.0.0 start
/Users/matteo/Repositories/js-best-practices-real-time-chat
> node ./index.js

Starting up
```

Volendo, possiamo aggiungere qualche informazione sulla nostra applicazione nel logging di avvio:

```
var package = require("./package");

console.log(package.name, "starting up");
console.log("version", package.version);
```

Possiamo persino inviare messaggi di log colorati usando il pacchetto `colors`. Per installarlo aggiungeremo l'opzione `--save`, che modificherà automaticamente il file `package.json`:

```
$ npm install colors --save
```

Al contenuto del file `package.json` verrà quindi aggiunto:

```
...
  "dependencies": {
    "colors": "~0.6.0-1"
  }
...
```

Possiamo quindi mostrare delle scritte colorate, in questo modo:

```
var package = require("./package");
var colors = require('colors');

var output = "";
output += package.name.red;
output += "@";
output += package.version.yellow;
output += " starting up".green;

console.log(output);
```

Potete trovare i sorgenti dell'applicazione sviluppata sinora all'indirizzo:

https://github.com/mcollina/js-best-practices-real-time-chat/tree/package.json

Maggiori opzioni per la libreria Colors sono disponibili all'indirizzo:

https://github.com/marak/colors.js/

Express

Il framework Express (http://expressjs.com/) consente di realizzare applicazioni web in Node.js astraendo dalla natura testuale del protocollo HTTP, ma allo stesso tempo consentendo di sfruttarne tutta la flessibilità. Per installarlo, eseguiamo:

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

Avviare un'applicazione Express è semplice:

```
var package = require("./package");
var colors = require('colors');
var express = require('express');
var app = express();

app.get("/", function (req, res) {
  res.send("Hello world!");
});

app.listen(3000, function (err) {
  if (err) {
    console.log(err);
```

```
    process.exit(1);
  }

  var output = "";
  output += package.name.red;
  output += "@";
  output += package.version.yellow;
  output += " started up on port 3000".green;

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

Dopo aver eseguito `npm start` possiamo puntare il nostro browser alla pagina http://localhost:3000/ e vedremo l'agognata scritta "Hello world!".
Express fornisce anche alcuni middleware molto utili che possiamo usare all'interno delle nostre applicazioni. Express si basa infatti su Connect http://www.senchalabs.org/connect/. All'interno di Express e Connect un middleware è un filtro e un'applicazione è composta da una serie di filtri in serie. Una richiesta HTTP può essere gestita da un filtro, in modo totale o parziale, ed eventualmente passarla al successivo.
I middleware più comuni sono:

- `express.static` per servire file statici;
- `express.logger` per stampare a video tutte le richieste HTTP che arrivano alla nostra applicazione;
- `express.bodyParser` per effettuare il parsing in automatico dei body delle richieste POST e PUT. Gestisce inoltre gli invii multipart (caricamento file) e la codifica JSON;
- `express.compress` per comprimere in GZIP tutti i file inviati ai client, al fine di ridurre il numero dei byte trasmessi;
- `express.cookieParser` effettua la lettura dei cookie;
- `express.cookieSession` realizza una sessione utente basata su cookie.

Unendoli tutti, avremo:

```
var package = require("./package");
var colors = require('colors');
var express = require('express');
var app = express();

app.use(express.logger());
app.use(express.compress());
app.use(express.bodyParser());
app.use(express.cookieParser());
app.use(express.cookieSession({
  secret: "a very long and RANDOM secret: 42"
}));
app.use(express.static("public"));
```

```
app.get("/", function (req, res) {
  res.send("Hello world!");
});

app.listen(3000, function (err) {
  if (err) {
    console.log(err);
    process.exit(1);
  }

  var output = "";
  output += package.name.red;
  output += "@";
  output += package.version.yellow;
  output += " started up on port 3000".green;

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

Per verificare che tutto funzioni, creiamo una cartella public all'interno del nostro progetto, poi inseriamo un file prova.txt con contenuto a piacere. Avviando l'app con il solito comando `npm start` possiamo vedere quanto aggiunto a `prova.txt` all'indirizzo http://localhost:3000/prova.txt. Potete vedere lo stato dei lavori all'indirizzo:

https://github.com/mcollina/js-best-practices-real-time-chat/tree/express-1

Bower

Bower (http://twitter.github.io/bower/) è un progetto Open Source di Twitter per promuovere lo sviluppo di componenti riutilizzabili per il web. Questo strumento è stato scritto in Node.js ed è facilmente installabile con un comando:

```
$ npm install bower -g
```

Bower consente di installare molti pacchetti compatibili con il semplice comando `bower`, che è estremamente simile a NPM. Infatti possiamo controllarne il funzionamento scrivendo nel file `component.json` le dipendenze della nostra applicazione:

```
{
  "name": "js-best-practices-real-time-chat",
  "version": "0.0.0",
  "dependencies": {
    "bootstrap": "~2.3.1",
    "font-awesome": "~3.0.2",
    "jquery": "~1.8.0"
  }
}
```

Eseguendo il comando `bower install` avremo tutte le dipendenze nella cartella `components/`. Al fine di costruire la nostra applicazione web, useremo all'interno del browser Twitter Bootstrap (http://twitter.github.io/bootstrap), Font-Awesome (http://fortawesome.github.io/Font-Awesome) e JQuery (http://jquery.com). L'applicazione allo stato attuale dei lavori è scaricabile all'indirizzo seguente:

https://github.com/mcollina/js-best-practices-real-time-chat/tree/bower

Grunt

Grunt (http://gruntjs.com) è uno strumento per automatizzare la concatenazione, la minificazione, l'esecuzione dei test unitari all'interno dei browser, il processo di verifica (tramite JsHint, www.jshint.com). Grunt è installabile tramite il semplice comando:

```
$ npm install grunt-cli -g
```

Inoltre, dobbiamo installarlo anche all'interno della nostra applicazione:

```
$ npm install grunt grunt-contrib-jshint grunt-contrib-concat \
  grunt-contrib-uglify grunt-contrib-watch grunt-contrib-less \
  --save
```

In particolare i vari pacchetti servono per:

- `grunt-contrib-jshint` - verificare i file JavaScript;
- `grunt-contrib-concat` - concatenare i file JavaScript;
- `grunt-contrib-uglify` - minificare i file JavaScript;
- `grunt-contrib-watch` - rieseguire i vari step ad ogni modifica,;
- `grunt-contrib-less` - compilare i file LESS (http://lesscss.org) in CSS (Bootstrap è un framework CSS scritto in LESS);
- `grunt-contrib-copy` - copiare i file binari, come per esempio le immagini o i font.

Ormai i lettori più smaliziati avranno capito che ogni strumento che introduciamo ha un suo file di configurazione, tipicamente scritto in JSON. Grunt non fa eccezione e consente di definire l'ordine in cui i vari task verranno eseguiti nel Gruntfile.js. Per quanto Grunt sia uno strumento basato su Node.js, nulla ci vieta di usarlo in ogni altro progetto web, per esempio basato su PHP.
La parte client della nostra applicazione userà jQuery e per semplicità tutti i file JavaScript di Twitter Bootstrap (che vedremo nella prossima sezione), più un file JavaScript scritto da noi. Il primo compito per cui useremo Grunt sarà concatenare questi file in un unico archivio, per poi scriverlo nella cartella `public/`. Le dipendenze, installate con

Bower, risiedono nella cartella `components/`, mentre scriveremo il nostro file JavaScript della cartella `assets/js`. Il `Gruntfile.js` per svolgere la concatenazione è il seguente:

```
module.exports = function(grunt) {

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {
      dist: {
        src: ['components/jquery/jquery.js','components/bootstrap/js/*.js', 'assets/js/*.js'],
        dest: 'public/js/main.js'
      }
    }
  });

  // Load from NPM packages
  grunt.loadNpmTasks('grunt-contrib-concat');

  // Default task(s).
  grunt.registerTask('default', ['concat']);
};
```

Possiamo notare come viene definita la configurazione della build, specificando per il task `concat` quali file concatenare. Eseguendo il comando `grunt`, otterremo in `public/js/main.js` il risultato della concatenazione.
Siccome vogliamo aggiornare la build in tempo reale, a ogni cambiamento dei file JavaScript, possiamo usare il plugin `grunt-contrib-watch`. Poiché l'abbiamo aggiunto al file `package.json`, configurarne il supporto all'interno di Grunt è facile:

```
module.exports = function(grunt) {

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {
      dist: {
        src: ['components/jquery/jquery.js','components/bootstrap/js/*.js', 'assets/js/*.js'],
        dest: 'public/main.js'
      }
    },
    watch: {
      src: {
        files: ['assets/**', 'components/**', 'Gruntfile.js'],
        tasks: ['default']
      }
    }
  });

  // Load from NPM packages
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-watch');
```

```
  // Default task(s).
  grunt.registerTask('default', ['concat']);
};
```

Eseguendo `grunt watch` in un terminale, a ogni nostra modifica dei file JavaScript o del `Gruntfile.js` stesso la build verrà rieseguita.
Dopo aver concatenato il file, è opportuno minificarlo se vogliamo eseguirlo in un ambiente di produzione. Il plugin `grunt-contrib-uglify` fornisce tutto il necessario per ridurre la dimensione del pacchetto così creato:

```
module.exports = function(grunt) {

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {
      dist: {
        src: ['components/jquery/jquery.js','components/bootstrap/js/*.js', 'assets/js/*.js'],
        dest: 'public/main.js'
      }
    },
    watch: {
      src: {
        files: ['assets/**', 'components/**', 'Gruntfile.js'],
        tasks: ['default']
      },
      options: {
        nospawn: true
      }
    },
    uglify: {
      dist: {
        src: '<%= concat.dist.dest %>',
        dest: 'public/js/main.js'
      }
    },
  });

  // Load from NPM packages
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // Default task(s).
  grunt.registerTask('default', ['concat']);

  // Production task
  grunt.registerTask('production', ['default', 'uglify']);
};
```

Nella definizione precedente possiamo notare come andiamo a definire un nuovo task `production`, che a valle di `default` (che a breve farà molto di più di una semplice concatenazione) compie anche la minificazione.

Poiché siamo bravi sviluppatori JavaScript, vogliamo automatizzare la verifica dei nostri programmi tramite JsHint (www.jshint.com per tutte le opzioni), in modo che non vi siano errori di sintassi al momento dell'esecuzione. Inoltre, possiamo usare lo strumento per definire una politica comune di formattazione del codice. Possiamo quindi eseguire:

```
module.exports = function(grunt) {

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {
      dist: {
        src: ['components/jquery/jquery.js','components/bootstrap/js/*.js', 'assets/js/*.js'],
        dest: 'public/main.js'
      }
    },
    watch: {
      src: {
        files: ['assets/**', 'components/**', 'Gruntfile.js'],
        tasks: ['default']
      },
       options: {
        nospawn: true
      }
    },
    uglify: {
      dist: {
        src: '<%= concat.dist.dest %>',
        dest: 'public/js/main.js'
      }
    },
    jshint: {
      all: ['assets/js/**/*.js'],
      options: {
        curly: true,
        eqeqeq: true,
        immed: true,
        latedef: true,
        newcap: true,
        noarg: true,
        sub: true,
        undef: true,
        unused: true,
        boss: true,
        eqnull: true,
        browser: true,
        globals: {
          console: true
        }
      }
    }
  });
```

```
  // Load from NPM packages
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-jshint');

  // Default task(s).
  grunt.registerTask('default', ['concat', 'jshint']);

  // Production task
  grunt.registerTask('production', ['default', 'uglify']);
};
```

Dal momento che Twitter Bootstrap è un framework CSS basato su LESS, Grunt ci consente di automatizzare la compilazione dei file LESS grazie al plugin `grunt-contrib-less`. Font-Awesome, il set di icone Open Source che useremo nella nostra applicazione, fornisce un file LESS utilizzabile assieme a Bootstrap. Possiamo quindi scrivere il file `assets/css/main.less` nel seguente modo:

```
@import "bootstrap.less";
@import "font-awesome";
```

Al fine di compilare il file `main.less`, modifichiamo nel seguente modo il `Gruntfile.js`:

```
module.exports = function(grunt) {

  // let do not repeat the css paths twice
  var cssPaths = ["components/bootstrap/less", "components/font-awesome/less",
"assets/css"];

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {
      dist: {
        src: ['components/jquery/jquery.js','components/bootstrap/js/*.js', 'assets/js/*.js'],
        dest: 'public/main.js'
      }
    },
    watch: {
      src: {
        files: ['assets/**', 'components/**', 'Gruntfile.js'],
        tasks: ['default']
      },
       options: {
        nospawn: true
      }
    },
    uglify: {
      dist: {
        src: '<%= concat.dist.dest %>',
        dest: 'public/js/main.js'
      }
```

```
    },
    jshint: {
      all: ['assets/js/**/*.js'],
      options: {
        curly: true,
        eqeqeq: true,
        immed: true,
        latedef: true,
        newcap: true,
        noarg: true,
        sub: true,
        undef: true,
        unused: true,
        boss: true,
        eqnull: true,
        browser: true,
        globals: {
          console: true
        }
      }
    },
    less: {
      development: {
        options: {
          paths: cssPaths
        },
        files: {
          "public/css/main.css": "assets/css/main.less"
        }
      },
      production: {
        options: {
          paths: cssPaths,
          yuicompress: true
        },
        files: {
          "public/css/main.css": "assets/css/main.less"
        }
      }
    },
  });

  // Load from NPM packages
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-less');

  // Default task(s).
  grunt.registerTask('default', ['concat', 'jshint', 'less:development']);

  // Production task
  grunt.registerTask('production', ['default', 'uglify', 'less:production']);
};
```

Nello snippet di codice precedente possiamo vedere come sia usata una funzionalità importante di Grunt, ossia la possibilità di specificare dei "target"; nel caso specifico di LESS questi sono `less:development` e `less:production`. In questo modo lo stesso plugin può essere eseguito con due configurazioni differenti.
Infine, poiché Font-Awesome richiede la presenza dei file per i font, useremo il plugin `grunt-contrib-copy` per copiare gli asset. Quindi:

```
module.exports = function(grunt) {

  // let do not repeat the css paths twice
  var cssPaths = ["components/bootstrap/less", "components/font-awesome/less", "assets/css"];

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {
      dist: {
        src: ['components/jquery/jquery.js','components/bootstrap/js/*.js', 'assets/js/*.js'],
        dest: 'public/main.js'
      }
    },
    watch: {
      src: {
        files: ['assets/**', 'components/**', 'Gruntfile.js'],
        tasks: ['default']
      }
    },
    uglify: {
      dist: {
        src: '<%= concat.dist.dest %>',
        dest: 'public/js/main.js'
      }
    },
    jshint: {
      all: ['assets/js/**/*.js'],
      options: {
        curly: true,
        eqeqeq: true,
        immed: true,
        latedef: true,
        newcap: true,
        noarg: true,
        sub: true,
        undef: true,
        unused: true,
        boss: true,
        eqnull: true,
        browser: true,
        globals: {
          console: true
        }
      }
    },
```

```
    less: {
      development: {
        options: {
          paths: cssPaths
        },
        files: {
          "public/css/main.css": "assets/css/main.less"
        }
      },
      production: {
        options: {
          paths: cssPaths,
          yuicompress: true
        },
        files: {
          "public/css/main.css": "assets/css/main.less"
        }
      }
    },
    copy: {
      main: {
        files: [{
          expand: true,
          flatten: true,
          src: ['components/font-awesome/font/*'],
          dest: 'public/font/'
        }]
      }
    }
  });

  // Load from NPM packages
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-less');
  grunt.loadNpmTasks('grunt-contrib-copy');

  // Default task(s).
  grunt.registerTask('default', ['concat', 'jshint', 'less:development', 'copy']);

  // Production task
  grunt.registerTask('production', ['default', 'uglify', 'less:production']);
};
```

Per supportare la compilazione automatica dei template JSON, possiamo integrare Grunt all'interno di Express, in modo che l'applicazione ricompili automaticamente i nostri file JavaScript e CSS in sviluppo. Aggiungiamo quindi al file `index.js`:

```
require("./Gruntfile")(grunt);

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

```
app.configure("development", function () {
  grunt.tasks(["default", "watch"]);
});

app.configure("production", function () {
  grunt.tasks(["production"]);
});
```

L'applicazione allo stato attuale dei lavori è scaricabile all'indirizzo:

https://github.com/mcollina/js-best-practices-real-time-chat/tree/grunt

Layout e templating in Express

È venuto il momento della redazione delle pagine web vere e proprie, ed è quindi opportuno vedere come Express consenta di specificare il layout della nostra applicazione. Infatti, Express permette di scrivere le proprie applicazioni usando diversi framework di templating. I più popolari sono EJS (Embedded JavaScript, http://embeddedjs.com) e Jade (http://jade-lang.com).
In questo esempio useremo Jade, un linguaggio particolare ma allo stesso tempo molto popolare, perché è quello consigliato dagli autori di Express. In ogni caso, la procedura per installare questi linguaggi è la medesima. Ovviamente, tutto inizia con un:

```
$ npm install jade --save
```

Inserire il supporto per Jade è semplicissimo. Basta aggiungere le seguenti linee di codice a index.js:

```
app.configure(function () {
  app.engine('jade', require('jade').__express);
  app.set('views', __dirname + '/views');
});
```

Occorre quindi predisporre il layout e l'unica pagina che useremo in questo momento. In particolare, entrambi questi file staranno nella cartella `views/`.
All'interno del file `layout.jade` inseriamo:

```
!!!
html
  head
    title JS Best Practice Example Chat
    meta(name='viewport', content='width=device-width, initial-scale=1.0')
    link(rel='stylesheet', href='/css/main.css')
  body
    block content

    script(src='/js/main.js')
```

Jade consente di ereditare il layout fra template e questo avviene grazie alla presenza dell'elemento `block content` (https://github.com/visionmedia/jade#template-inheritance), che ci consente di specificare all'interno di `views/index.jade`:

```
extend layout

block content
  p Hello world
```

Il progetto fino a questo punto è disponibile all'indirizzo:

https://github.com/mcollina/js-best-practices-real-time-chat/tree/jade

Bootstrap

Bootstrap fornisce componenti HTML, CSS e JavaScript per la realizzazione veloce di applicazioni web. In particolare, nasce come framework CSS ed è scritto in LESS, ma contiene anche alcuni plugin JavaScript che dipendono da JQuery. Infine, consente lo sviluppo di applicazioni responsive.
Il primo componente che la nostra chat utilizza è la barra di navigazione. Per installarla, modifichiamo il file `views/layout.jade` nel modo seguente:

```
!!!
html
  head
    title JS Best Practice Example Chat
    meta(name='viewport', content='width=device-width, initial-scale=1.0')
    link(rel='stylesheet', href='/css/main.css')
  body
    .navbar.navbar-fixed-top
      .navbar-inner
        .container
          a.brand(href="/") JS Best Practice Example Chat
          ul.nav
            li
              a(href="http://www.matteocollina.com") About

    block content

    script(src='/js/main.js')
```

Inoltre, occorre modificare il file `views/index.jade` in modo che adotti le convenzioni di stile di Bootstrap:

```
extend layout

block content
  .container
    .row
```

```
      .span12
        p Hello world
```

È poi necessario aggiustare il padding del body, in modo da non avere testo "sotto" alla navbar, perciò occorre cambiare il file `assets/css/main.less` nel modo seguente:

```
@import "bootstrap.less";
@import "font-awesome";

body {
  // this is needed to shift down to allow the
  // navbar to stay at the top
  padding-top: 60px;
}
```

Quindi, aggiungiamo il form di invio all'interno di `views/index.jade`:

```
extend layout

block content
  .container
    .row
      form
        .span1
        input#name.span2(type="text", placeholder="Your name")
        .input-append
          input.span8.appendedInput#message(type="text", placeholder="Your text")
          button#send.btn(type="button") Send
```

Per visualizzare i messaggi useremo una tabella:

```
extend layout

block content
  .container
    .row
      form
        .span1
        input#name.span2(type="text", placeholder="Your name")
        .input-append
          input#message.span8.appendedInput(type="text", placeholder="Your text")
          button#send.btn(type="button") Send

    .row
      hr

    .row
      .span12
        table#messages.table.table-striped
          thead
            tr
              td Name
              td Message
          tbody
```

```
tr
  td Fullo
  td Ciao Matteo!
tr
  td Matteo
  td Ciao Fullo!
```

Infine possiamo aggiungere alcune icone dal set di Font-Awesome e migliorare il risultato finale, che potete osservare nella Figura 7.1 e all'URL:

https://github.com/mcollina/js-best-practices-real-time-chat/tree/bootstrap

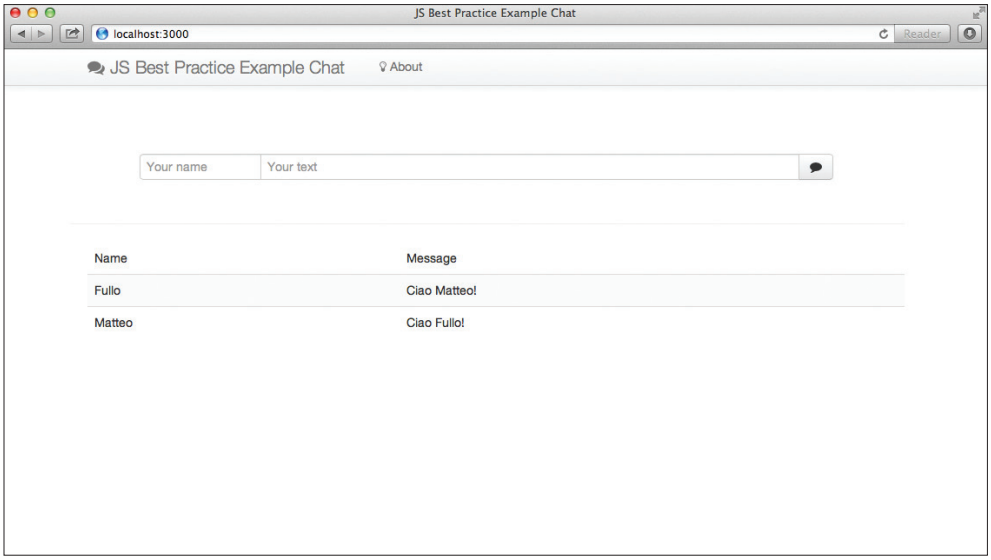

Figura 7.1 - L'interfaccia utente della nostra chat basata su Bootstrap e le icone Font-Awesome.

Socket.io

Socket.io (http://socket.io) è una delle librerie più usate per mostrare la semplicità di utilizzo di Node.js. Infatti, realizzare la stessa cosa in Ruby, Python, Java o C# richiede molto più lavoro. Installare socket.io è, come sempre, semplicissimo:

```
$ npm install socket.io --save
```

È poi necessario modificare il file `index.js` e aggiungere all'applicazione Express il supporto a Socket.io. Per farlo, occorre sostituire la linea:

```
app.listen(3000, function (err) {
```

con:

```
var server = require("http").createServer(app);
io = io.listen(server);

io.sockets.on('connection', function (socket) {
  console.log("new client..".green, "feels good to be loved!".yellow);
});

server.listen(3000, function (err) {
```

Questa modifica è necessaria perché Socket.io va installato direttamente sul server HTTP, e non sull'app Express. Il codice, quindi, va a creare un server HTTP specificando l'app Express, per poi procedere all'installazione di Socket.io. Infine, tramite la chiamata a `server.listen`, mettiamo in ascolto il server HTTP sulla porta 3000.
L'installazione di Socket.io non si limita al server, ma va compiuta anche all'interno delle pagine web. In particolare, bisogna aggiungere in coda al file `views/layout.jade` il file JavaScript di Socket.io:

```
    script(src='/socket.io/socket.io.js')
```

e modificare il file `assets/js/chat.js` nel modo seguente:

```
$(window).ready(function () {
  var socket = io.connect('http://localhost');

  socket.on('connect', function () {
    // la socket è connessa
  });
});
```

Per avere una notifica sullo standard output del server, aggiungiamo:

```
io.sockets.on('connection', function (socket) {
  console.log("new client..".green, "feels good to be loved!".yellow);
});
```

Ora, se si avvia il server e si apre un browser alla pagina http://localhost:3000, vedremo che un nuovo client si collega al nostro server ed è pronto a ricevere messaggi. L'app a questo stadio di sviluppo è disponibile all'indirizzo:

https://github.com/mcollina/js-best-practices-real-time-chat/tree/socket.io-1

Implementare la chat richiede la modifica del file `index.js`, rimpiazzando la callback per l'evento `connection` con:

```
io.sockets.on('connection', function (socket) {
  console.log("new client..".green, "feels good to be loved!".yellow);
```

```
  socket.on("chat", function (message) {
    socket.broadcast.emit("chat", message);
  });
});
```

Queste poche righe di codice trasformano ogni messaggio chat che proviene da un client in un messaggio che verrà inviato a tutti i client connessi (broadcast). Il broadcast non viene però inviato al client che l'ha scatenato.
Per mostrare sia i messaggi che arrivano sia quelli generati dal client corrente è opportuno creare gli elementi HTML tramite un template Underscore. Quindi si modifica il file `views/index.jade`, aggiungendo:

```
  script#message-template(type="text/template")
    tr
      td <%- username %>
      td <%- message %>
```

Occorre anche installare Underscore (`bower install underscore --save`) e aggiungerlo al `Gruntfile.js`. Per mostrare il messaggio all'interno dell'applicazione web modifichiamo il file `assets/chat.js`:

```
$(window).ready(function () {
  var socket = io.connect('http://localhost');
  var form = $("form");
  var username = $("#username");
  var message = $("#message");
  var messages = $("#messages");
  var messageTemplate = _.template($("#message-template").html());

  var update = function (data) {
    var element = $(messageTemplate(data));
    element.hide().fadeIn('slow');
    messages.prepend(element);
  };

  form.submit(function () {
    var data = { username: username.val(), message: message.val() };
    update(data);
    return false;
  });
});
```

Per inviare i messaggi, occorre modificare il file `assets/chat.js`:

```
$(window).ready(function () {
  var socket = io.connect('http://localhost');
  var send = $("#send");
  var form = $("form");
  var username = $("#username");
  var message = $("#message");
  var messages = $("#messages");
```

```
  var messageTemplate = _.template($("#message-template").html());

  var update = function (data) {
    var element = $(messageTemplate(data));
    element.hide().fadeIn('slow');
    messages.prepend(element);
  };

  send.attr("disabled", "disabled");

  socket.on('connect', function () {
    send.removeAttr("disabled");
  });

  socket.on("chat", update);

  form.submit(function () {
    var data = { username: username.val(), message: message.val() };
    socket.emit("chat", data);
    update(data);
    return false;
  });
});
```

È importante notare sia il metodo `socket.emit` sia `socket.on`. Il primo viene usato per inviare un messaggio di tipo chat al server Socket.io. Il secondo consente di specificare una callback che verrà chiamata alla ricezione di ogni messaggio di tipo chat dal server. Per verificare che tutto funzioni come atteso basta aprire due browser su http://localhost:3000 e inviare alcuni messaggi. Il risultato finale è disponibile all'indirizzo:

https://github.com/mcollina/js-best-practices-real-time-chat/tree/socket.io-2

Conclusioni

In questo capitolo abbiamo trattato Node.js, il framework che consente di eseguire JavaScript server-side. Abbiamo trattato le librerie più diffuse e abbiamo visto come usarle per risolvere problemi reali, come minificare JavaScript e CSS, effettuare richieste HTTP, rispondere a richiese HTTP, costruire una semplice chat in tempo reale. Per brevità l'accesso ai database non è stato affrontato, ma Node.js ha librerie per accedere sia a database relazionali, come PostgreSQL e MySQL, sia noSQL, come MongoDB, Redis, Riak o OrientDB.

Node.js può essere usato sia al fianco di altre tecnologie, come PHP, Ruby, Python, Java o C#, sia come base tecnologica per un nuovo progetto. Molte società lo hanno adottato per risolvere specifici problemi tecnologici, come per esempio LinkedIn, AirBnB e Wallmart. Alcune startup sono invece interamente basate su Node.js, come Storify, Klout e Geekli.st.