Node.js

Rev 6.1 del 10/09/2021

Introduzione a Node.js

E' la base di una piattaforma denominata ***MEAN*** (Mongo, Express, Angular, **Node**) mirata alla realizzazione di **applicazioni web di tipo client server** interamente basate sul linguaggio JavaScript, arricchito con diverse funzionalità server side.

Mongo = database
Express = web server
Angular = client side
Node = server side

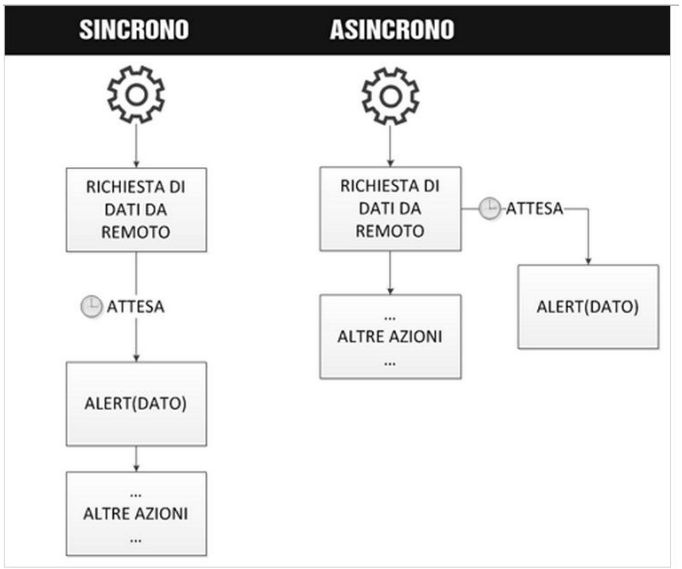
E' basato sul **JavaScript Engine V8**, che è una piattaforma open source che gira su tutti i principali SO. La scelta di edificare Node.js "**sopra**" V8 è una garanzia di **sicurezza**, **portabilità** e **stabilità.**

Caratteristiche:

- Completamente basato sulla programmazione asincrona
- **Ottime prestazioni** (sia su server di piccole dimensioni sia su server di grandi dimensioni)
- Possibilità di interfacciamento con i **socket** a livello 4 della pila ISO OSI

Nodejs gestisce tutte le principali attività in modo asincrono, sfruttando al massimo l'approccio event-loop tipico di javascript.

In figura è rappresentata la differenza tra l'approccio sincrono e l'approccio asincrono nella gestione di una certa richiesta.



In node.js tutte le principali attività vengono gestite in modo asincrono. Ad esempio

- **L'accesso alle risorse del sistema operativo** (come ad esempio l'acceso ad un dbms o l'accesso ad un file), in modo molto diverso rispetto ai web server classici in cui ogni operazione viene trattata in **modo sincrono** eventualmente sospendendo il thread durante l'attesa
- **La gestione delle richieste di rete** in cui ogni richiesta viene gestita mediante una istanza di una apposita funzione di callback associata alla richiesta stessa.

  Anche questo è profondamente diverso rispetto ai web server tradizionali in cui il server esegue un ciclo di attesa su una specifica porta http e, per ogni richiesta, viene recuperato un thread da un pool di thread in attesa, che si occuperà di gestire la comunicazione con il client. Nella programmazione multithread tradizionale i vari thread vengono messi "in wait" tra una richiesta all'altra, e di fatto rimangono nell'elenco dei processi "idle" da gestire da parte del sistema operativo.

Distribuzione ed Installazione

La prima release di Node.js risale al 2009 a cura dalla società californiana Joyent.
L'installazione di Node.js è molto semplice. L'applicazione viene installata in **C:/Programmi/Nodejs** e, nelle versioni più recenti, viene creato un **Nodejs command prompt** che gestisce in automatico alcune variabili d'ambiente relative alle librerie installate in modo globale all'interno del profilo utente.
A dicembre 2020 l'ultima versione stabile e con supporto garantito a lungo termine (**LTS** = long term support) è la versione **14**.

Esecuzione

Il comando **node** senza parametri apre un terminale che consente di interagire a linea di comando con node.js. Ad esempio il comando `c:> console.log("hello world");`
chiede a node di stampare a video la stringa "`hello world`".

Il comando **node –v** consente di visualizzare la versione di Note correntemente installata.
Digitando da terminale **where node** si può vedere il percorso completo di installazione di node

Normalmente però Node viene lanciato passandogli come parametro sulla linea di comando il nome di un file testuale.js contenente le istruzioni da eseguire: **node filename.js**

L'impostazione del path **set PATH = %PATH%;C:\Program Files\nodejs**
nelle ultime versioni fatto in automatico dall'installer

Richiami sulle HTTP request

Le richieste che un server web può ricevere contengono sempre **tre** informazioni fondamentali:

- La **risorsa richiesta**, che può essere una pagina HTML, oppure un file CSS o JS, oppure un insieme di dati in formato JSON o XML.
  Es di richiesta: **www.vallauri.edu/orario** dove www.vallauri.edu rappresenta il nome del web server mentre **orario** rappresenta la risorsa richiesta. Se non viene indicata nessuna risorsa il browser invia un semplice / che solitamente viene interpretato dal server come richiesta della home page, cioè index.html

- Eventuali **parametri** concatenati in coda alla risorsa e separati tramite un punto interrogativo
  Es di richiesta: `www.vallauri.edu/orario?classe=5B`

- Il **metodo di chiamata**, che può essere GET / POST / PATCH / PUT / DELETE
  - In caso di richieste GET eventuali parametri vengono concatenati in modo visibile in coda alla URL
  - In caso di richieste POST PATCH PUT DELETE eventuali parametri vengono passati in forma nascosta all'interno del body della http request. Formalmente non vi è nessuna differenza fra queste quattro modalità. Esse offrono semplicemente al server la possibilità di rispondere in quattro modi diversi a seconda del tipo di chiamata
  - Nel caso delle web form dotate di un pulsante di submit, il contenuto di tutti i controlli presenti nella web form viene automaticamente inviato al server sotto forma di parametri, concatenati alla URL o in forma nascosta a seconda del tipo di chiamata

Principali status code ritornati dalle http response

Un web server può rispondere ad una HTTP request con uno dei seguenti codici di errore:

- **Error 404 Page Not Found** Significa dominio esistente, ma risorsa non trovata
- **Error 401 Unauthorized** Significa credenziali non valide
- **Error 403 Forbidden** (credenziali ok ma accesso alla risorsa non consentito ad esempio in caso di token non valido o scaduto)
- **Error 500 Internal Server Error** Significa che si è verificato errore nel codice lato server
- **Error 503 Database Connection Error**
- **Error 400 Bad Request** Il server non riesce ad interpretare la richiesta.
  Ad esempio per un **errore sintattico** nei parametri, a causa di un parametro mancante oppure di un valore atteso numerico ma in realtà non numerico
- **Error 422 Unprocessable Entity** The server understands the content type of the request entity and the syntax of the request entity is correct (thus a 400 (Bad Request) status code is inappropriate) but was unable to process the contained instructions. Ad esempio per un **errore semantico** nei parametri come ad esempio un parametro fuori valore, ad esempio eta < 18

Il primo esercizio

Realizziamo un semplice web server che restituisca al client una pagina HTML contenente le tre informazioni ricevute in fase di richiesta, cioè risorsa richiesta, parametri e metodo di chiamata.
Il codice è basato sul metodo **`http.createServer`** che crea e restituisce un server HTTP. Il metodo createServer si aspetta come parametro una **funzione di callback** (indicata come **RequestListener**) **che sarà eseguita in corrispondenza di ogni richiesta ricevuta** dal server. In corrispondenza di ogni richiesta, alla funzione di callback verranno automaticamente iniettati due parametri:

- **request**, un oggetto di tipo `http.IncomingMessage` che rappresenta il messaggio HTTP ricevuto dal client contenente tutte le informazioni relative alla richiesta
- **response**, un oggetto di tipo `http.ServerResponse` all'interno del quale il server dovrà andare a scrivere la risposta da restituire al client

```
let _http = require("http");
let _url = require("url");
let port = 1337;

let server=_http.createServer(function (req, res) {
     let metodo = req.method;

     // parsing della url ricevuta dal client
     let url = _url.parse(req.url, true);
     let risorsa = url.pathname;
     let param = url.query;
     let dominio = req.headers.host;

     res.writeHead(200,{"Content-Type": "text/html;charset=utf-8" });
     res.write("<h1> Informazioni relative alla Richiesta ricevuta</h1>");
     res.write("<br>");
     res.write(`<p> Risorsa richiesta : ${risorsa} </p>`);  // alt 96
     res.write(`<p> Metodo : ${metodo}</p>`);
     param = JSON.stringify(param)
     res.write(`<p> Parametri : ${param}</p>`);
     res.write(`<p> Dominio richiesto : ${dominio}</p>`);
     res.end();

     let colors = require("colors");
     console.log("Richiesta Ricevuta : + url.path.yellow ); // risorsa + parametri
});

// se non si specifica l'indirizzo IP di ascolto il server viene avviato su tutte le interfacce
server.listen(port);
console.log("server in ascolto sulla porta " + port);
```

Per provare il funzionamento del web sever aprire un browser e digitare
`http://localhost:1337/orario?classe=5B` Per terminare il server digitare **CTRL + C**

La funzione di callback di createServer( ) può anche essere scritta esternamente come named function:

```
var server = http.createServer(rispondi)
function rispondi(request, response) {
}
```

La soluzione anonima ha il **vantaggio** che la funzione di callback può 'vedere' tutte le variabili della procedura in cui si trova.

La soluzione tramite named function è invece vantaggiosa quando la funzione di callback deve essere richiamata in punti diversi. Può comunque accedere all'oggetto in cui si trova (nell'esempio oggetto **server**) utilizzando la parola chiave **this**.

Avvio e debug di una applicazione node tramite Visual Studio Code

Digitando **code .** dal prompt della cartella di lavoro corrente, si apre direttamente Visual Studio Code sul progetto corrente. In alternativa si può aprire Visual Studio Code dal desktop e selezionare il comando **File / OpenFolder** .

Per avviare l'applicazione occorre selezionare il file principale (server.js) **Run / Start Debugging** oppure **Start Without Debugging.**

In entrambi i casi dal menù contestuale che compare selezionare **Node.js** oppure, nelle versioni più vecchie, **Node.js(preview**)

I moduli nativi di Node.js

Il modulo http

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

Ritorna un oggetto di tipo **http.Server.** Riceve come parametro una **funzione di gestione delle richieste http** che verrà avviata **in un thread dedicato** in corrispondenza di ogni singola richiesta.

Questa funzione presenta due parametri che riceverà dal server al momento dell'evento:

- **request** (di tipo **http.IncomingMessage)** contenente le informazioni relative alla richiesta
- **response** (di tipo **http.ServerResponse** su cui la funzione di callback andrà a scrivere la risposta da restituire al clienr

NOTA: Nelle connessioni persistenti (default in HTTP 1.1 - `Connection:Keep-Alive`), l'evento request può essere generato sul server più volte all'interno della stessa connessione (sembra che venga generato sistematicamente 2 volte)
In ogni caso, dopo che è stato invocato il metodo **end(),** nulla viene più inviato al browser.

http.get (url, function (response) { }
Metodo che consente di inviare una richiesta http get ad un web server (tramite un http server che può aver bisogno di interpellare un altro server per avere certe informazioni)
Il **primo parametro** rappresenta la url da contattare, che può essere espressa sia come url parsificata, sia come stringa. E' anche possibile passare soltanto l'**host** (come stringa o parsificato)
Il **secondo parametro** rappresenta una funzione di callback che verrà eseguita in corrispondenza del ricevimento della risposta e che presenta come parametro un oggetto di tipo http.IncomingMessage.

L'oggetto `http.Server`

Oggetto ritornato da create Server(). Presenta i seguenti metodi :

`server.listen``(port, [hostname], [backlog], [callback])`

`port` = porta di ascolto del server HTTP
`hostname` = indirizzo ip dell'interfaccia di ascolto.
`backlog` = dimensione max della coda delle connessioni pendenti. Il valore di default è 511
`callaback` = funzione eseguita in corrispondenza dell'evento `listening`

- Se il server viene avviato su **127.0.0.1** sarà accessibile **solo** impostando 127.0.0.1 sul browser
- Se invece viene avviato sull'**IP della macchina**, sarà accessibile dal browser **solo** impostando l'IP della macchina (non 127.0.0.1). Sarà inoltre accessibile anche dall'esterno.
- **Se hostname viene omesso** (`INADDR_ANY`), il server accetterà connessioni su tutte le interfacce.

`server.close``([callback])` arresta il server

L'oggetto RESPONSE

Oggetto che rappresenta la risposta http da inviare al richiedente a seguito di una http request

`response.setHeader(``'Content-Type', 'text/plain'`**`);`**
Consente di impostare le intestazioni http. Può essere richiamato più volte, una per ogni intestazione.

`response.writeHead``(statusCode, [headers]);`
Chiude la scrittura delle intestazioni e le invia al client insieme allo status code.
Deve essere richiamato una sola volta prima dell'invio della risposta.

`statusCode` Codice a 3 cifre che rappresenta lo stato della risposta **200 = OK**
`headers` = collezione di intestazioni separate da virgola (alternativo a **`setHead`**)

```
var msg="Hello World"
var headers = {'Content-Length':msg.length,
               'Cache-Control':'no-cache',
               'Content-Type':'text/plain;charset=utf-8'}; (con trattino!)
response.writeHead (200, headers);
```

*Nell'esempio **`content-length`** indica il numero di bytes e non il numero di caratteri.*
Nel caso di risposte testuali funziona comunque perché ogni carattere occupa un solo byte, altrimenti occorrerebbe Buffer.byteLength(msg).

`response.write``(data, [encoding]);`
consente di inviare il body della pagina. Può essere richiamato più volte
Ritorna `true` se l'intero buffer è stato inviato correttamente
`data` può essere essere un buffer di bytes oppure una stringa
`encoding` indica il formato della risposta (default **`utf8`**). Attenzione che in questo caso utf8 si scrive **SENZA trattino**. Trattandosi però del valore di default, può SEMPRE essere omesso.

`response.end``([data],[encoding]);`
Avvisa il client che i dati sono terminati (intestazioni e body).
DEVE essere esplicitamente richiamato al termine di ogni response.
I parametri, se specificati, sono equivalenti a quelli di response.write.
Non termina l'esecuzione dello script, che continua con le istruzioni successive. Se però le righe successive contengono altre istruzioni di invio dati al client, l'invio non viene eseguito.

`response.redirect``("/page2.html");`

L'oggetto REQUEST

Contiene le informazioni relative alla httpRequest appena ricevuta. Presenta le seguenti proprietà:

request.method Restituisce come stringa il Request Method ('get' o 'post').
request.url E' la URL testuale impostata dall'utente. Dalla versione 4 in avanti contiene soltanto il **path** (risorsa e parametri) e non l'**host** (hostname e porta) che sono diventati campi di **req.headers**
request.headers Restituisce l'elenco delle headers come vettore associativo.
Ad esempio **req.headers.host** contiene il nome del dominio e la porta utilizzate dal client

Il modulo url

Contiene alcune comode funzioni per la manipolazione delle URL.
Espone il metodo statico **url.parse** che converte url testuali in un corrispondente JSON:

```
var url = require('url');
var _url = url.parse(request.url, true);
```

- **Passando false** (default) **i parametri vengono restituiti come stringa e la proprietà .query restituirà a sua volta una stringa**
- **Passando true la funzione esegue un parsing completo della url, comprensivo anche dei parametri, sia che essi siano url-encoded, sia json.** Il problema è che, nel secondo caso, se i parametri sono GET, il browser aggiunge i caratteri speciali della codifica url-encoded

Supponendo che **request.url** contenga '*ipoteticamente*' la seguente stringa

'http://user:pass@www.host.com:8080/a/t/h/pagina1.html?nome=valore#hash'

l'oggetto _url esporrà le seguenti proprietà (le prime sette non più presenti come detto ad inizio pagina)

```
href: The full URL that was originally parsed. Protocol and host are lowercased
  Example:  'http://user:pass@www.host.com:8080/a/t/h/èagina1.html?nome=valore#hash'
protocol: The request protocol, lowercased.
  Example:  'http:'
slashes: The protocol requires slashes after the colon (colon = due punti)
  Example:  true
auth: The authentication information portion of a URL.
  Example:  'user:pass'
hostname: Just the lowercased hostname portion of the host.
  Example:  'www.host.com'
port: The port number portion of the host.
  Example:  '8080'
host: The full lowercased host portion of the URL (hostname:port)
  Example:  'www.host.com:8080'

pathname: The path section of the URL, that comes after the host and before the
  queryString, including the initial slash if present. Example:  '/p/a/t/h'
search: The 'query string' portion of the URL, including the leading question mark.
  Example:  '?nome=valore
query: Restituisce la 'query string' portion of the url, in formato di stringa
'nome=valore' se il seconda parametro è FALSE (default) oppure come object
{'nome':'valore'} se il secondo parametro è TRUE
path: Concatenation of pathname and search.
  Example:  '/p/a/t/h?nome=valore
hash: The 'fragment' portion of the URL including the pound-sign. Example:  '#hash'
```

Una URL può essere scritta direttamente in formato JSON :

```
{ hostname:'www.html.it',   protocol:'http',   port:8080 }
```

Altri metodi statici dell'oggetto url

`url.format` riceve una url parsificata `{ }` e restituisce la url completa come stringa
`url.resolve` consente di modificare una url come indicato nell'esempio.

Esempi

```
var url = require('url');
var parsed = url.parse('http://www.html.it:8080');
// visualizza tutte le singole property
for (e in parsed) console.log(parsed[e]);

console.log(url.format(
    { hostname:'www.html.it', protocol:'http', port:8080 } ));
url.resolve('/one/two/three', 'four')         // '/one/two/four'
url.resolve('http://example.com/', '/one')    // 'http://example.com/one
url.resolve('http://example.com/one', '/two') // 'http://example.com/two'
```

Il modulo FS – File System

Permette di leggere e scrivere risorse nel file system del server, eseguendo tutte le operazioni tipiche di questo ambito come ad esempio la copia, la lettura. la scrittura. la cancellazione di files e cartelle.

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

Il metodo `readFile()` legge l'intero file ricevuto come parametro.

```
fs.readFile(path, 'utf8', function (err, data){});
```

Parametri:

- il **path** del file da leggere
- il tipo di **encoding**. Il file viene restituito come **`raw buffer`**, cioè come buffer binario nudo e crudo. Il parametro encoding consente di convertire il raw buffer nella codifica indicata, che può essere "**`utf8`**" per il testo, "**`base64`**" per immagini base64, "`binary`" per files binari.
  Nel caso di immagini occorre omettere il parametro encoding e trasmettere il buffer cos'ì com'è
  Il 2° parametro in realtà può sempre essere omesso. In caso di file testuale può essere visualizzato semplicemente facendo **data.toString().**
- una funzione di **callback** che dovrà essere eseguita al termine della lettura del file e che riceve due parametri:
  - un oggetto **`err`** che è null in caso di successo oppure settato in caso di errore
  - un oggetto **`data`** contenente i dati desiderati.

Esempio:

```
fs.readFile('./myFile', function (err, data) {
     if (err) console.log('Error');
     else     console.log(data.toString());  });
```

Il metodo **`readFileSync`** è analogo al precedente ma sincrono, cioè bloccante fino al termine della lettura

```
var data = fs.readFileSync("./myFile", "utf8");
fs.existsSync("./myFile");
```

Restituisce `true` se il file esiste, altrimenti `false`

Nota: I metodi del modulo **fs**, per accedere al file system, utilizzano le stesse notazioni di Linux, per cui il path può essere espresso :

- come percorso relativo che deve iniziare con il nome del file oppure con **puntoslash `./`**
- come percorso assoluto, accessibile tramite __dirname (che è l'analogo di **Server.MaphPath** di ASP) a cui occorre concatenare il nome del file preceduto dal solo **slash** (senza puntino)

Al di fuori di **fs**, tutti gli altri moduli non hanno esigenza di accedere al file system, per cui non 'vedono' oltre la web directory di lavoro. **Per loro / rappresenta la web root direcotory.**

Istruzioni per la scrittura su un file di testo

```
var fs = require('fs');
fs.writeFile("./tmp/file.txt", "Hey there!", function(err) {
    if(err) {
        console.log(err);
    } else {
        console.log("The file was saved!");
    }
});
```

per appendere in coda ad un file si può utilizzare il seguente metodo che presenta la stessa sintassi del precedente:

```
fs.appendFile(("./tmp/file.txt", 'data to append', function (err) {

});
```

Nota sulla restituzione di una immagine

Nel momento in cui ajax aggiunge dinamicamente una immagine all'interno del DOM, **automaticamente** il browser provvede a richiedere al server la risorsa indicata.

- Nel caso di apache, il server provvede automaticamente ad inviare la risorsa richiesta, per cui l'intero processo è completamente trasparente
- Nel caso invece di node.js / espress, occorre definire esplicitamente un listener che si occupi di individuare la risorsa attraverso il file system, leggerla tramite readFile ed inviarla al client tramite response.end (oppure definire un listener statico di gestione delle risorse statiche).

Il modulo util

Questo modulo contiene funzioni per la formattazione di date, stringhe, funzionalità di debug e altre utilità, come ad esempio l'introspezione delle variabili. Esempi di Metodi:

- una serie di funzioni logiche **`is*`** (per esempio **`isArray`** o **`isDate`**)
- **`format`** per la formattazione di stringhe a partire da placeholder
- **`debug`** e **`log`** per il monitoraggio del flusso.
- **`inspect`** restituisce la rappresentazione testuale di un oggetto. Dopo il nome dell'oggetto sono possibili altre 4 opzioni che però di solito sono omesse accettando il valore di default.
  (In molti casi la conversione viene comunque eseguita automaticamente).

Il modulo globals

Il modulo globals è un modulo atipico in quanto non rappresenta un vero e proprio oggetto ma una serie di API incluse nel namespace globale dell'applicazione e quindi direttamente accessibili senza nessuna inclusione. Tra le funzioni di questo pseudo-modulo occorre ricordare:

L'oggetto **`console`** permette di accedere a standard output e a standard error del processo
Il metodo **`require`** consente di includere moduli aggiuntivi e ritorna un riferimento al modulo.
Il metodo **`module`** consente di esporre unoggetto o singoli metodi rendendoli disponibili tramite require

La variabile **`__filename`** contiene il nome del file.js avviato in esecuzione
La variabile **`__dirname`** contiene il percorso assoluto della cartella corrente (dove si trova file.js)

npm

Node Package Manager è un **Gestore di Pacchetti** java script (installato automaticamente insieme a node js) che gestisce il **repository npmjs.com** (simile a github) dove sono disponibili migliaia di moduli aggiuntivi javascript, fra i quali anche jquery e bootstrap. Chiunque può pubblicare sul repository npm un modulo di sua produzione che ritiene possa essere riutilizzato da altre persone. Analogo a **apk-get** di ubuntu.

Per scaricare un nuovo modulo aprire un terminale e scrivere semplicemente

```
npm install moduleName          // se già esiste non fa nulla
npm install moduleName@latest   // aggiorna all'ultima versione
npm install moduleName@5.4.1    // aggiorna alla versione specifica
```

il modulo verrà installato **all'interno della cartella corrente** in una sottocartella **node_modules**.
Per utilizzare il modulo all'interno di una applicazione è sufficiente richiamarlo tramite il metodo globale

```
require('moduleName')  // anche senza .js
```

Il metodo **require** provvede a ricercare il modulo indicato sulla base del seguente schema :

1. fra i moduli nativi di Nodejs.
2. all'interno di una cartella **node_modules** posizionata all'interno della attuale cartella di lavoro
3. se non lo trova provvede a ricercare la cartella **node_modules** all'interno di tutte le cartelle genitrici della cartella corrente, fino eventualmente alla cartella root
4. Se non lo trova lo cerca nel path indicato all'interno della variabile di ambiente **NODE_PATH**

In alternativa è anche possibile (ma ovviamente sconsigliato) indicare un path completo assoluto

```
require('C:/mia Cartella/node_modules/mioModulo);
```

oppure un path completo relativo:

```
require('./mia Cartella/mioModulo);
```

Installazione globale nel profilo utente

L'opzione **–g** (--global) fa sì che il modulo venga installato a livello globale all'interno di una cartella legata all'utente **C:/Users/userName/AppData/Roaming/npm/node_modules/**

Questo path viene automaticamente aggiunto alla variabile di ambiente **PATH** in fase di installazione, per cui tutte le utility eseguibili da riga di comando (es cordova.exe) possono essere individuate ed eseguite dal launcher di windows. Nelle versioni precedenti occorreva aggiornare manualmente la variable PATH

Per quanto riguarda invece il **require** dall'interno di una applicazione nodejs, se l'applicazione viene eseguita tramite il '*nodejs command prompt'* i moduli globali vengono individuati automaticamente.
Se invece si utilizza un '*prompt normale'* i moduli installati globalmente non vengono visti ed occorre configurare manualmente la variabile di ambiente NODE_PATH.

NODE_PATH → `C:\Users\userName\AppData\Roaming\npm\node_modules`

Il file `package.json`

A monte della cartella **node_modules** dove npm installa tutti i moduli relativi al progetto corrente, è normalmente presente un file **package.json** che viene utilizzato per fornire a npm :

- indicazioni sul progetto (nome, descrizione, etc)
- e soprattutto **l'elenco dei moduli utilizzati dal progetto**, ciascuno con il proprio numero di versione. Questo elenco rappresenta le **dipendenze** del progetto, cioè l'elenco dei moduli aggiuntivi necessari per il funzionamento del progetto.

Una delle funzionalità del file package.json è quella per cui, se si pubblica l'applicazione su un repository, quando qualcuno scaricherà la nostra applicazione tramite npm, **tutte le dipendenze indicate verranno anch'esse automaticamente scaricate** dal repository npmjs indicato all'inizio.

Il comando npm init

Il comando **npm init** (sostanzialmente analogo a ***git init***) provvede ad inizializzare la cartella di lavoro come cartella NPM creando il file **package.json.**
Prima di creae il file il comando richiede di inserire alcune informazioni sul tipo di progetto:

```
"name": "esercizi_nodejs",        // solo caratteri minuscoli
"version": "1.0.0",               // major.minor.patch
"description": "my npm library",
"entry point": "server.js",       // main file
"keywords": node, web, server
"author": "roberto.mana  <ing.mana@tiscali.it> (http://robertomana.it)",
"license": "ISC"
"dependencies": {
    "express": "^4.14.0",
    "mongoose": "^4.7.2"
}
"repository" : {"type":"git", "url":"git://github.com/documentcloud/myProject.git"},
"homepage" : "http://documentcloud.github.com/myProject/"
```

Note:

- Come **name** di default viene proposto il name della cartella corrente dalla quale è stato lanciato **npm init**. Se il nome di questa cartella contiene degli spazi, la procedura va in errore !!
- Il segno **^** nelle dipendenza significa **compatibile con** (cioè di versione >= a quella indicata).
- La **licenza ISC** (**Internet Systems Consortium)** consiste nei segueti permessi:
  Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies. Cioè è concesso il permesso di utilizzare, copiare, modificare e / o distribuire questo software per qualsiasi scopo con o senza corrispettivo, a condizione che l'avviso di copyright di cui sopra e l'avviso di autorizzazione compaiano in tutte le copie.

Il comando npm install

- Una volta creato il file package.json tutte le volte che si installa una nuova libreria il file viene automaticamente aggiornato. Viceversa i comandi **npm install** (o **npm upgrade**) **utilizzati senza parametri provvedono ad installare in locale** (nella sottocartella node_modules del progetto corrente) tutte le dipendenze indicate all'interno del file **package.json** **non ancora presenti nel progetto**.
  La differenza consiste nel fatto che **npm install**, in caso di libreria già esistente, non fa nulla mentre **npm upgrade** controlla anche le versioni, e se la libreria richiesta è presente sulla macchina ma in una versione più vecchia rispetto a quanto indicato nel file packege.josn, la aggiorna automaticamente.
  Il comando **npm uninstall -g packName** rimuove il package indicato.
  Il comando **npm install packName@9.0.0** installa la versione indicata.
  Il comando **npm install packName latest** installa l'ultima versione (default).
  Il comando **npm cache clean -f** ripulisce la cache prima di nuove installazioni

Le sottodipendenze

- Anche i moduli a loro volta possono avere delle dipendenze che verranno automaticamente installate insieme al modulo medesimo (leggendole nel file package.json del modulo stesso).
- Quando si aggiungono dei moduli manualmente tramite `npm install moduleName`, se si specifica l'opzione **--save,** npm provvede a scaricare anche le dipendenze e ad aggiornare il file **package.json.** Da npm5 in avanti queste operazioni vengono eseguite in automatico senza il --save.
- All'interno di package.json è anche possibile aggiunegere un campo **devDependencies** (**development dependencies),** , cioè dipendenze che NON andranno a fare parte della build finale ma che sono necessarie in fase di sviluppo. Per aggiornare l'elenco delle devDependencies occorre utilizzare l'opzione **npm install --dev**

npm si tiene una cache, per cui se si ripete npm install la volta successiva è molto più veloce.

Il file `package-lock.json`

Contiene un maggiore dettaglio sulle dipendenze del progetto corrente e viene usato per scaricare le dipendenze riassunte nel file **`package.json`**. Le installazioni recenti utilizzano entrambi i files.

Creazione ed esposizione di un nuovo modulo

In un'architettura modulare e scalabile può risultare conveniente scrivere **nuovi moduli** che poi saranno riutilizzabili nelle varie applicazioni. Un modulo è un file che definisce una serie di funzioni o oggetti JavaScript e che le espone attraverso il metodo **module.exports**.

Se i moduli si trovano nella sottocartella **`node_modules`** della cartella cartella vengono visti scrivendo semplicemente:

```
require('modulo');
```

Se invece si trovano direttamente nella cartella di lavoro (senza la sottocartella **`node_modules`**) occorre specificare il path assoluto relativo:

```
require('./modulo.js');
```

Esempio 1 : Codice scritto direttamente

```
// modulo.js
    console.log("Codice scritto direttamente");

// main.js
    require('modulo.js');
```

Il codice scritto direttamente viene eseguito nel momento stesso del require(), senza necessità di utilizzare il metodo module.exports

Esempio 2 : Export di una funzione in forma anonima

```
// modulo.js
    module.exports = function () {
        console.log("Funzione Anonima");
    };
// oppure
    function _anonima (){
        console.log("Funzione Anonima");
    };
    module.exports = _anonima;

// main.js
    var mod = require('modulo.js');
    mod();
```

Esporre una funzione in forma anonima significa esporla senza riassegnargli un nome specifico.
Il chiamante può richiamare la funzione anonima utilizzando semplicemente il nome della variabile con ()

- **Un modulo può esporre una unica funzione anonima.**
- **In caso di export di più funzioni anonime, l'ultima 'nasconde' tutte le precedenti**
- **L'eventuale export di una funzione anonima deve essere fatto prima degli export delle funzioni esplicite.**

Esempio 3 : Export di funzioni esplicite

```
// modulo.js
    function _somma(a, b) {
       return a + b;  }

    var _moltiplicazione = function(a, b) {
       return a * b;  }

    module.exports = _somma;
    module.exports.molptiplicazione = _moltiplicazione;
```

Esporre una funzione in forma esplicia significa esporla riassegnargli un nome specifico. Nell'esempio
_somma viene esportata in modo anonimo e 'nasconde' eventuali export precedenti.
_ moltiplicazione viene invece esportata in modo esplcito con un suo nome.

```
// main.js
    var mod = require('modulo.js');
    console.log(mod(5,2));
    console.log(mod.moltiplicazione(5,2));
```

Esempio 4 : Export di un JSON

```
// modulo.js
var _json = {
     nome:"pippo",
     setNome:function(s) { this.nome = s }
};
```

Notare che i json possono anche contenere dei metodi, i quali per poter accedere alle Properties devono utilizzare la parola chiave **this.** (Se si omette il this all'interno della function, essendo var facoltativo, **nome** viene interpretato come nuova variabile locale e NON come property)

Come per le funzioni, anche gli object possono essere esposti in forma **anonima** o in forma **esplicita**:

Esposizione anonima:

```
module.exports = _json;

// main.js
var mod = require('modulo.js');
console.log(mod.nome);
mod.setNome("pluto");
console.log(mod.nome);
```

Esposizione esplicita:

```
module.exports.json = _json;

// main.js
var mod = require('modulo.js');
console.log(mod.json.nome);
mod.json.setNome("minnie");
console.log(mod.json.nome);

// oppure
var json = require('modulo.js').json;
console.log(json.nome);
```

Export di una classe con sintassi ES5

Invece di definire un semplice json come il precedente, è possibile in javascript ES5 definire una classe nel modo seguente:

```
// modulo.js
var _myClass = function(){
    this.nome="pippo";
    this.setNome=function(s){this.nome=s};
}

var myClass =  new _myClass();
module.exports.myClass = myClass;
```

Notare che la classe istanziata (myClass) è **in tutto e per tutto equivalente al json dell'esempio precedente** per cui,le istruzioni di accesso ed utilizzo della classe sono le STESSE IDENTICHE dell'esempio precedente (del json).

E' anche possibile, ovviamente, esportare la 'classe' e poi eseguire l'istanza all'interno del modulo principale:

```
module.exports.myClass = _myClass;  // cioè senza il new

// main
var mod = require('modulo.js');
var myClass = new mod.myClass;
console.log(myClass.nome);
```

Export di una classe con sintassi ES6

```
class _myClassES6 {
    constructor() {}
    // property
    nome,
    cognome,
    // method
    setNome(s) { this.nome = s }
}
myClass ES6 = new _myClassES6()
module.exports.myClassES6 = _myClassES6  // esporto la classe
module.exports.myClassES6 = myClassES6   // esparto la classe già istanziata
```

Import di un file json

E' anche possibile importare direttamente un file con estensione **.json** contenente un singolo json:

data.json

```
{
  "name": "Freddie Mercury"
  "band": "Queen"
}
```

main.js

```
var obj = require('data.json');
console.log(obj.name);  //Freddie Mercury
```

Concetto di Dispatcher

Il dispatcher rappresenta l'interfaccia di `front-end` di un web server, cioè il modulo che si prende in carico le **Http Request** e ritorna al chiamante le corrispondenti **Http Response.**
In italiano significa letteralmente **`centralino`** o **`smistatore`**, cioè un componente che permette di differenziare le chiamate effettuate dall'utente, effettuando la selezione sulla base della risorsa richiesta

In node.js il dispatcher può essere richiamato all'interno della funzione di callback del metodo **`createServer`**() dell'oggetto **`http`** con l'injection dei parametri request e resposne.
Per ogni richiesta ricevuta, il Dispatcher dovrà prendere in considerazione tre informazioni :

- **Metodo usato per la richiesta**
- **Risorsa richiesta**
- **Eventuali Parametri aggiuntivi**

Lettura del Metodo di chiamata

Occorre utilizzare la proprietà **`method`** dell'oggetto Request:

```
var method = req.method
```

Lettura della Risorsa e dei Parametri GET

Occorre utilizzare la proprietà testuale **`url`** dell'oggetto Request:

```
    var _url = require('url').parse(req.url, false)
    var risorsa = _url.pathname;
    var params = _url.query;   // sotto forma di stringa
```

Esattamente come in PHP, se più variabili hanno lo stesso nome (es **`chkHobby`**) l'ultimo valore sovrascrive i precedenti. Se invece il nome comune è vettoriale (es **`chkHobby[]`**), la proprietà .query restituisce all'interno di un unico parametro (avente nome sempre **`chkHobby[]`**) tutti i valori presenti separati da una **virgola**.

Struttura della classe Dispatcher:

```
var dispatcher = function() {
    this.prompt = ">> >> ";
    this.listeners = {
        GET: {},
        POST: {},
        DELETE: {},
        PUT: {},
        PATCH: {}
    };
}
```

La proprietà **`listeners`** è preposta a contenere un elenco di ***`listeners`*** che dovranno servire le varie richieste che possono arrivare al server.
Ogni ***`listener`*** è costituito un json avente come **chiave** la risorsa da servire, e come **valore** una funzione di callback da eseguire in corrispondenza della richiesta di quella risorsa.

```
{ "risorsa1": callback1,
  "risorsa2": callback2,
  "risorsa3": callback3,
}
```

I ***listeners*** vengono **suddivisi** in più vettori associativi: il vettore dei listener GET, il vettore dei listener POST, un vettore dei listener DELETE e così via. Questa suddivisione è una scelta del tutto arbitraria dettata dalla modalità di lavoro di express.

In realtà in un'ottica **CRUD** sarebbe meglio tenere tutti i listener **raggruppati** insieme (indipendentemente dal metodo). In questo modo per ogni risorsa il server potrà gestire un unico listener che eseguira uno switch del tipo:

```
case GET :
case POST :
```

```
dispatcher.prototype.addListener = function(risorsa, metodo, callback) {
     metodo = metodo.toUpperCase();
     this.list[metodo][risorsa]=callback;
}

dispatcher.prototype.showList = function(){
      console.log("Elenco dei Listener registrati:")
      for(var key in this.list) {
            console.log(key); // "GET" - "POST"
            var listeners=this.list[key];
            for(var key in listeners ) {
                  console.log(key);
            }
      }
}

dispatcher.prototype.dispatch = function(req, res) {
    var metodo = req.method.toUpperCase();

    // Lettura dei parametri GET (intercettati SEMPRE, qualunque sia il method)
    var _url = url.parse(req.url, false)
    var risorsa = _url.pathname;
    var params = _url.query

    // parametri GET in formato url-encoded
    req["GET"] = require("querystring").parse(params);

    // parametri GET in formato JSON
    params = decodeURIComponent(params);
    //req["GET"] = JSON.parse(params)

    // log
    console.log(this.prompt + metodo + ":" + risorsa);
    if (metodo != "GET")
      console.log("  parametri " + metodo + ":" + JSON.stringify(req[metodo]));
    console.log("   parametri get:" + JSON.stringify(req["GET"]));

    // Dispatch
    if (risorsa in this.listeners[metodo]) {
        var callback = this.listeners[metodo][risorsa];
        callback(req, res);
    } else
        this.staticListener(req, res)
}

// export in forma anonima
module.exports = new dispatcher();
```

Note

- Il metodo invocato (GET o POST) viene di solito restituito dal browser in **MAIUSCOLO**.
- Il metodo **`dispatch`** provvede a ricercare ed eseguire il listener corrispondente alla richiesta ricevuta. Riceve i soliti due parametri **Request** e **Response** e li passa alla funzione di callback
- La funzione di **callback** relativa ai listener potrebbe avere qualsiasi firma decisa dal programmatore. Dovendo però costruire la risposta da inviare al client, dovrà ricevere almeno il parametro **`response`** sul quale andare a scrivere la risposta. Si passano di solito **`req`** e **`res`**

Passaggio dei parametri come stringa json

$.ajax() consente di passare i parametri al server in formato url-encoded oppure in formato json (parametro **`contentType`**).

- Impostando come formato **url-encoded** è possibile passare a $.ajax() anche oggetti json che però non possono essere espressi sotto forma di vettore e non possono contenere al loro interno altri oggetti annidati. $.ajax() provvedde automaticamente a convertirli in url-encoded,
- Impostando invece come formato **json**, diventa possibile utilizzare vettori ed oggetti annidati. In questo caso i parametri devono essere passati a $.ajax() come **stringa json.** Lato server sarà quindi sufficiente eseguire un parsing dei parametri

Nel caso dei parametri GET (che comunque di solito vengono passati in **url-encoded**) il passaggio dei parametri come **stringa json** comporta una ulteriore complicanza perchè, nel caso della URL, il browser provvede ad eseguire la cosiddetta URL encode, cioè provvede a codificare I caratteri non standard (come ad esempio gli spazi presenti nella stringa json) con la codifica esadecimale urlEncode, per cui il server, prima del parsing, dovrà eseguire la corrispondente decodifica:

```
var s = url.parse(req.url, false).query;
s = decodeURIComponent(s);
```

*Nel caso di json annidati è possibile utilizzare il metodo **`$.param(obj)`** che esegue la conversione da json a url-encoded applicando anche la codifica esadecimale urlEncode.*

Restituzione di risorse statiche

Per le richieste relative a pagine statiche, è sufficiente cercare la risorsa all'interno del File System e restituirla. Occorre però capire se la risorsa richiesta è statica oppure dinamica. A tal fine si possono seguire diversi approcci:

- Supporre che le richieste di servizio inizino tutte con **`/api`**. In tal caso il test `if(risorsa.substr(0,4)=="/api")` indica la richiesta di un servizio
- Utilizzare per i files lato server una **estensione specifica** (come nel caso di .php e .asp). L'estensione potrebbe essere **.njs** che però non è riconosciuta dagli editor
- Scorrere prima i vettori relativi ai listener dinamici e, se non si trova un listener associato, si interpreta la richiesta come richiesta di una risorsa statica e si cerca il file all'interno del file system. Se il file non viene trovato si restituisce un errore.

Nel metodo **`dispatch`**() precedente si è scelta **la terza soluzione**, dove `staticListener` è un metodo che restituisce al client la risorsa statica richiesta (se esiste)

```
dispatcher.prototype.staticListener = function(req, res){
    var risorsa = url.parse(req.url).pathname;
    if (risorsa == '/')
        risorsa = "/index.html";
```

```
    var filename = "./static" + risorsa;

    var _this = this;
    // filename = percorso relativo o assoluto tramite __dirname

    fs.readFile(filename, function(err, content) {
        if (err)
            _this.errorListener(req, res);
        else {
            var header = { "Content-Type": mime.getType(filename) }
            res.writeHead(200, header);
            res.end(content);
        }
    });
}
```

- Il modulo **mime** (installato a parte) consente di accedere al content-type del file.
- All'interno di **readFile** **this** corrisponde a **fs**, per cui per poter richiamare correttamente errorListener occorre prima salvare il riferimento all'interno di una variabile temporanea.

La gestione degli errori

Se il client richiede una risorsa inesistente e non si gestiscono gli errori, il client rimane 'appeso' fino allo scadere di un timeout. Per la gestione degli errori occorre differenziare

- nel caso della richiesta di risorse statiche, in caso di risorsa non trovata, occorre restituire una pagina **error.html**
- nel caso della richiesta di risorse dinamiche, in caso di risorsa non trovata, occorre restituire una semplice stringa da visualizzare lato client in caso di fail

```
dispatcher.prototype.errorListener = function(req, res){
    var risorsa = url.parse(req.url)["pathname"];
    if (risorsa.substr(0, 4) == "/api") {
        var header = { "Content-Type": 'application/json;charset=utf-8' };
        res.writeHead(404, header);
        res.end('"Risorsa non trovata"');
    } else {
        var header = { "Content-Type": 'text/html;charset=utf-8' };
        res.writeHead(404, header);
        res.end(stringaErrore);
    }
}

dispatcher.prototype.init = function(req, res) {
    fs.readFile("./static/error.html", function(err, content) {
        if (err)
            content = "<h1>Risorsa non trovata</h1>"
        stringaErrore = content.toString();
    });
}
```

L'oggetto err

Presenta le seguenti proprietà principali:

- **code** codice dell'errore
- **stack** stack completo dei messagi errre (visualizzato di default)
- **message** Ultimo messaggio di errore in cima allo stack (user error)

Gestione dell'errore lato server

```
function error(req, res, err,
            code_error = 500, string_error = 'Internal Server Error') {
   // Log dell'errore restituito da node / mongoDB
   console.log(err.code + ' ' + err.message);
   res.writeHead(code_error, HEADERS.TEXT);
   res.end(string_error);
}
```

Gestione dell'errore lato client

Quando riceve un codice diverso da 200, il metodo **$.ajax()** richiama automaticamente l'evento **fail()** all'interno del quale si può utilizzare il seguente codice:

```
function error(jqXHR, testStatus, strError) {
    if (jqXHR.status == 0)
        alert("Connection Refused or Server Timeout");
    else if (jqXHR.status == 200)
        alert("Errore Formattazione dati\n" + jqXHR.responseText);
    else
        alert("Server Error: " + jqXHR.status + " - " + jqXHR.responseText);
}
```

Main di prova della classe Dispatcher e registrazione dei listener

```
var http = require('http') ;
var dispatcher = require('dispatcher.js');

var server = http.createServer(function (req, res) {
     dispatcher.dispatch(req, res) ;
});
server.listen(1337, 'localhost', dispatcher.init());

dispatcher.addListener("/paginaGet.html", "GET", function(req, res) {
    res.writeHead(200, header);
    var data = "benvenuto " + req.GET["nome"]
    res.end(data);
});

dispatcher.addListener("/api/risorsa1", "GET", function(req, res) {
    res.writeHead(200, header);
    var data = { "benvenuto ": req.GET["nome"] }
    res.end(JSON.stringify(data));
});

dispatcher.addListener("/paginaPost.html", "POST", function(req, res) {
    res.writeHead(200, header);
    var data = "benvenuto " + req.POST["nome"]
    res.end(data);  });
```

La lettura dei parametri POST DELETE PUT PATCH

La lettura dei parametri POST è molto più complessa rispetto alla lettura dei parametri GET.
I parametri POST, PUT, PATCH e DELETE vengono infatti inviati al server all'interno del body della HTTP request ed occorre andare a leggerli manualmente,

Per poter accedere al body della HTTP request occorre sfruttare due eventi esposti dall'oggetto http.ServerRequest:

- **`data`**, che viene generato più volte, ad ogni chunk di dati ricevuto
- **`end`**, che è scatenato solo una volta, al completamento della ricezione

Il metodo `parsePostParameters()`

```
dispatcher.prototype.parsePostParameters = function(req, res) {
    var body = "";

    req.on("data", function(data) {
        body += data;
    });

    let _this = this;
    req.on("end", function() {
        // parametri url-encoded :
        var parametri = require("querystring").parse(body);

        // parametri json :
        // var parametri = JSON.parse(body)

        var metodo = req.method.toUpperCase();
        req[metodo] = parametri;
        _this.innerDispatch(req, res);
    });
);
```

- Il metodo **`querystring.parse()`** è simile a url.parse(), ma si aspetta come parametro la SOLA querystring
- All'oggetto **`req`** viene aggiunta una nuova property denominata **`POST`** e contenente tutti i parametri POST già parsificati
- Nel caso di modalità **DELETE PUT PATCH** i parametri vengono passati nel body della HTTP request esattamente come avviene per i parametri POST e possono essere letti allo stesso modo

Modifica del metodo dispatch()

C'è però ancora un altro problema. Nel momento in cui il dispatcher intercetta la richiesta e manda in esecuzione il listener corrispondente, i parametri post NON sono ancora stati intercettati, per cui risulterebbero UNDEFINED. Pertanto occorre posticipare l'esecuzione del dispatcher solo **DOPO** che i parametri post siano stati ricevuti ed acquisiti.

```
if(request.method.toUpperCase() == "GET")
     this.innerDispatch(req, res);
else if(request.method.toUpperCase() != "GET")
    this.parsePostParameters(req, res);
```

Soluzione alternativa

Un'altra soluzione consiste nel conglobare il richiamo al metodo **dispatch** (nel main) all'interno di una ulteriore funzione di callback richiamata SOLTANTO quando l'evento end ha terminato di leggere i parametri post. Il metodo **createServer** del main deve essere modificato nel modo seguente:

```
var server = http.createServer(function(req, res){
      if(request.method.toUpperCase() == "POST")
           dispatcher.parsePostParameters(req, res, function(){
                dispatcher.dispatch(req, res);
           });
     else if(request.method.toUpperCase() == "GET")
           dispatcher.dispatch(request, response);  });
```

Attenzione però che se a **dispatcher.parsePostParameters** si passa come terzo parametro direttamente **this.dispatch** in forma esplicita, viene fuori un Run Time Error che dice "**callback is not a function**". Questo perché come funzione di callback occorre passare una function priva di parametri ma non è ammesso passare il metodo di una classe.
Notare infatti come la funzione di callback sia PRIVA di parametric. request e response sono quelli della riga sopra dove request è stato arricchito con I parametric POST

Visione delle cartelle

- Nel **codice del server,** in ogni file, il path corrente è quello da cui è stato lanciato il createServer
- Il **require** opera secondo i criteri di npm, iniziando la ricerca dalla cartella corrente e risalendo l'intero albero verso l'alto. In alternativa è possibile specificare il path assoluto o relativo.
- Le **pagine HTML** richiedono i loro documenti (CSS, JS, etc) a partire dalla posizione in cui si trova la pagina html sul server. Se la pagina HTML si trova all'interno di una sottocartella, il browser provvede automaticamente ad aggiungere davanti al nome dei file il path con tutte le cartelle necessarie. Questo NON vale invece per **Ajax**, in cui il richiedente deve indicare ogni volta esplicitamente il path completo del servizio richiesto, a partire dalla cartella di lancio del server
- **JSDOM** anche jsdom opera attraverso il **file system** caricando il file jquery.js a partire dalla posizione corrente da cui è stato lanciato il createServer. La particolarità del metodo jsdom.env() è però quella di aggiungere un link alla libreria jQuery in coda alla pagina html da inviare al browser:
  `<script class="jsdom"  src="./jquery.js"> </script>`
  per cui il browser, ricevuta la pagina html, provvede ad inviare una ulteriore richiesta al server richiedendo la pagina jQuery.js ed anteponendo il path relativo utilizzato per accedere alla pagina html. Se però il browser, relativamente alla libreria jQuery, effettua una richiesta esplicita antecedente, la richiesta inserita da jsdom viene ignorata.

Debug tramite il modulo node-inspector

Si tratta di un **debugger** avanzato che sfrutta il debugger del browser.
Deve essere installato in modalità globale:

```
npm install node-inspector
```

Occorre quindi avviare su due terminali separati:

```
node --debug-brk myProgram
      debugger listening on port 5858
node-ispector
      visit http://127.0.0.1:8080/debug?port=5858 to start debugging
```

Aprire quindi nel browser l'indirizzo indicato. Si apre una finestra con il debugger del browser che consente di debuggare tramite browser una applicazione server !

Il modulo async

L'oggetto fornisce un utile supporto alla programmazione asincrona. Si supponga di dover eseguire due operazioni indipendenti sui dati di un database , e di dover restituire un risultato al client quando entrambe siano teminate. Le due funzioni, essendo indipendenti, potrebbero essere scritte una sotto l'altra. Essendo asincrone, node ne lancia una e, immediatamente dopo, lancia anche l'altra. Si pone però il problema di DOVE SCRIVERE le istruzione che ritornano il risultato al client. Nella callback della prima oppure nella callback della seconda ? Essendo asincrone è impossibile prevedere quale delle due finirà per prima. Le possibile soluzioni sono due:

- Scrivere le due chiamate in modo annidato, cioè scrivere la seconda chiamata al termine della callback della prima. Funziona benissimo ma si ricade nella programmazione sincrona (la seconda attende la prima).
- Al termine di ciascuna delle due callback si incrementa un medesimo contatore. Quella che trova il contatore a 2 invia i dati al client. Funziona benissimo anche questa ma diventa difficile da gestire nel caso di più operazioni parallele, con fastidiose ripetizione di blocchi di codice. Che cosa capita poi se uno dei due blocchi va in errore ?

Il metodo async.parallel

Ecco che interviene in questo caso l'oggetto async con il suo metodo **`.parallel`** che si aspetta due parametri:

- Un vettore di funzioni (task), che verranno tutte eseguite in parallelo, ciascuna delle quali si aspetta come parametro una funzione di callback che dovrà essere richiamata al termine dell'elaboorazione
- Una funzione di callback da eseguire alla fine quando tutti i task hanno eseguito il loro richiamo alla funzione di callback. Se anche uno solo dei task ha richiamato la sua callback passandogli err, la funzione finale riceverà `err != null` e segnalerà l'errore. Se nessuno dei task ha attivato err, la funzione finale troverà `err==null` e provvederà ad eseguire le operazioni successive.

```
async.parallel([
   function(callback) {
        db.save('a', function(err, data) {
            callback(err, data);   // se err è null passa null
        });
   },
   function(callback) {
        // il puntatore alla callback può essere passato direttamente a dbsave
        db.save('b', callback);
   }
 ],
function(err, data) {
    if (err)
        res.end("Errore Query");
    else
        res.end(JSON.stringify(data[1]));
});
```

Il parametro **`data`** iniettato alla funzione finale è un vettore enumerativo in cui :

- nella cella 0 sono contenuti i **`data`** della prima funzione
- nella cella 1 sono contenuti i **`data`** della seconda funzione

In realtà nella chiamata delle callabck il parametro `data` può anche essere omesso.
In tal caso il parametro finale `data` sarà ovviamente null.

In alternativa è anche possibile passare ad async.parallel un **vettore associativo** invece che enumerativo:

```
async.parallel({
   "one": function(callback) {
        db.save('a', function(err, data) {
            callback(err, data);
        });
   },
   "two": function(callback) {
        db.save('b', callback);
   }
},
function(err, jsonData) {
    if (!err)
       res.end(JSON.stringify(jsonData["two"]));
});
```

Il parametro jsonData sarà ora un JSON con tutti i risultati restituiti dai vari task:

```
{"one": data1,  "two": data2}
```

Il metodo async.series

Questo metodo interviene quando i task sono interdipendenti fra loro, cioè il secondo non può essere lanciato fino a quando il primo non è terminato. In questo caso async.series equivale alla scrittura dei task in modo annidato. Quando i task aumentano, async.series risulta molto più leggibile rispetto alla scrittura annidata e soprattutto semplifica notevolmente la gestione degli errori.

La sintassi è la stessa identica di prima, con un vettore di funzioni ed una callback finale che riceve come parametro err. La seconda funzione del vettore viene richiamata quando la prima termina richiamando la propria funzione di callback senza passargli `err`. Se la prima termina passando `err` alla funzione di callback, la seconda non viene richiamata ma viene richiamata immediatamente la callback finale che riceverà come parametro `err==true`.

Il metodo async.forEach

Ricevuto come parametro un vettore enumerativo di json (collezione), consente di iterare sulla collezione lanciando **in parallelo** un task asincrono per ogni elemento della collezione. Esegue una callback finale quando tutti i task sono terminati.

```
async.forEach(items, task, callback)
```

- **`items`** è la collezione su cui si vuole iterare (**vettore enumerativo di object**)
- **`task`** è la funzione da eseguire in parallelo per ogni item
- **`callback`** è la funzione fnale richiamata quando tutti i task sono terminati. Riceve un parametro **`err = true`** in caso di errore su uno dei task, **`= false`** se tutti sono terminati senza errore

Nell'esempio si suppone che data sia un vettore enumerativo di json restituiti da una query precedente:

```
async.forEach(data,
              function(item, callback ){
                   if (elab is ok) callback(null);
                   else if(err)   callback(err)
              },
              function(err) {
                   if(!err) inviaRisposta(req, res);
              });
```

Il modulo bind

il modulo **bind** consente di creare dinamicamente una pagina web caricando la struttura da un file TPL e valorizzandolo tramite javascript. Il nome **bind** indica un collegamento dinamico ad un file esistente.

I files TPL

Un file .tpl è un file **T**em**PL**ate (file modello), cioè un file che contiene soltanto il modello di come dovrà essere strutturata la pagina, senza contenere i dati veri e propri (esattamente come i moduli di Word). In questo modo l'utente potrà aggiungere dinamicamente da codice le informazioni per il completamento della pagina e creare più documenti senza dover ricreare ogni volta l'intera struttura della pagina. In un file .TPL le variabili che dovranno essere valorizzate run time devono essere scritte nel modo seguente :

```
(:nomeVariabile:)
```

E' anche possibile, tramite l'operatore **~,** assegnare alla variabile un valore di default che verrà assegnato al campo qualora, a run time, il chiamante non dovesse valorizzare il parametro.

```
<h1>(:nome ~ Marco:)</h1>
<b> (:indrizzo:) - (:citta:)</b>
```

Il metodo bind.toFile( )

Il metodo principale di bind è **toFile** che consente in un sol colpo di caricare il file template e di valorizzare tutti i parametri. Il metodo **toFile** prevede tre parametri:

- Il **path** del file TPL da caricare
- Un **javascript Object** contenente i valori da assegnare ai parametri del file TPL
- Una funzione di **callback** da eseguire al termine della creazione della pagina dinamica alla quale BIND passerà come parametro la pagina HTML costruita dinamicamente.
  Questa funzione viene di solito utilizzata per inviare la pagina al client.

```
var bind = require('./node_modules/bind');
dispatcher.addListener("/pageB", "get", function(req, res) {
    bind.toFile("./paginaDinamica.tpl",
    { nome:"Alby", indirizzo:"via Roma", citta:"milano" }, function(data){
            res.writeHead(200, { "Content-Length": data.length,
                                 "Content-Type": "text/html" });
            res.end(data, 'utf8'); }
    });
});
```

Utilizzo di una funzione java script per assegnare un valore ai parametri

Come secondo parametro del metodo **bind.toFile()**, anziché assegnare singoli valori ai parametri del file TPL, per ogni parametro è possibile assegnare un valore elaborato tramite una funzione javascript che può andare, ad esempio, a leggerlo su un DB.

```
dispatcher.addListener("/pageC", "get", function(req, res) {
    bind.toFile("./paginaDinamica2.tpl",
       {
            myPar: function(callback, a) {
            var ris = leggiDaDb() ;     // oppure
            var ris = parseInt(a)+1;
            return callback(ris);                }
       },
       function(data){
           res.writeHead(200, {"Content-Type": "text/html" });
           res.end(data, 'utf8');
       });
```

Notare **l'uso annidato** delle funzioni di callback.
All'interno del modulo BIND è definita una certa funzione che provvede a scrivere il dato del programma sul file template al termine del suo caricamento. Il puntatore a questa funzione viene passato come parametro di nome **`callback`** al metodo **`bind.toFile().`** Invece di assegnare un valore diretto ad un certo parametro, il programmatore può decidere di richiamare una funzione locale la quale, ad esempio, provvede a leggere una certa informazione da database e, **al termine**, dovrà richiamare la funzione callback passandogli come parametro il valore da inviare al client.

Non solo, ma la funzione javascript locale riceve anche un **secondo parametro**, che rappresenta l'eventuale valore di default del parametro **`myPar`** così come definito all'interno del template.

Invio di una response XML basata su Template

Anche una response XML può essere costruita sulla base di un file XML definito tramite Template.

Nell'intestazione della response occorre però impostare :
`"Content-Type": "text/xml"`

Costruzione del template

```
<?xml version="1.0" ?>
<person>
    <name>(:name:)</name>
    <city>(:city:)</city>
     <skills>
        (:skills ~
            <skill>
                <skill>[:skill:]</skill>
                <grade>[:grade:]</grade>
            </skill>
        :)
     </skills>
</person>
```

La scritta **`(:skills ~`** ..... **`:)`** indica che quello corrente non è un singolo parametro da valorizzare, ma un intero elenco di parametri di lunghezza imprecisata. Ogni elemento dell'elenco sarà costituito da due campi `[:skill:]` e `[:grade:]` racchiusi tra parentesi non più tonde ma quadre.

Codice di valorizzazione dei parametri

```
dispatcher.addListener("/pageXML", "get", function(req, res) {
    bind.toFile("./paginaXML.tpl",
      {
         name: 'Alberto',
         city: 'Milano',
         skills: [
            {  skill: 'NodeJS',  grade: 'A' },
            {  skill: 'JavaScript',  grade: 'A' },
            {  skill: 'Bash scripting',  grade: 'B' }
         ]
      },
      function(data){
            res.writeHead(200, { "Content-Length": data.length,
                                 "Content-Type": "text/xml" });
            res.end(data, 'utf8');
      });
});
```

Il modulo JSDOM ver 9.0.0

Valorizzare un file template non è comodissimo. Del resto il file HTML è un file testuale ed operare direttamente sulle stringhe sarebbe ancora peggio. Sono disponibili diversi moduli parser che ricevono come parametro una **stringa html** o una **pagina html** e restituiscono il **DOM** corrispondente, sul quale si può agire apportando modifiche, aggiungendo nodi, cancellando nodi, etc.

jsdom

Riceve una stringa html e restituisce il DOM corrispondente su cui sono resi disponibili i metodi jQuery. il modulo **jsdom**, è basato su Pyton 2.7, pertanto prima di installare il modulo occorre installare Pyton 2.7 (cartella c:\windows\pyton27). Poi si installa jsdom

```
npm install jsdom@9.0.0     // La versione 10 è profondamente cambiata
```

Dopo di che occorre ancora copiare la libreria jQuery all'interno della cartella di lavoro (sul server !), o meglio ancora all'interno della cartella node_modules della cartella di lavoro. Volendo è possibile anche aggiungere degli script jQuery alla pagina, per cui jsdom provvede automaticamente ad inserire all'interno della pagina un link alla libreria jQuery del server, la quale verrà quindi inviata al client mediante una send successiva, senza necessità di dover definire manualmente un apposito listener.

Utilizzo del metodo jsdom.env()

Il metodo **jsdom.env** è un metodo asincrono (che termina subito) e che si aspetta tre parametri:

- Il file html a cui si vuole associare il selettore jQuery
- Il path della libreria jquery all'interno del server
- Il **terzo parametro** è una funzione di callback verrà eseguita dopo aver caricato la libreria jQuery. Questa funzione di callback riceverà come parametri (da jsdom.env stessa) un puntatore all'oggetto **errors** (stack degli errori) ed un puntatore all'oggetto **window** corrente (finestra corrente), all'interno della quale **.$** rappresenta il selettore jQuery base. Per cui la funzione di callback potrà utilizzare il selettore jQuery $ per accedere e modificare i tag della pagina html.

Per evitare di dover richiamare ogni volta **jsdom.env()**, si può creare il seguente wrapper :

```
function caricaDom(myFile, callback) {
    jsdom.env(
        var page = fs.readFileSync(myFile);
        [ './jquery.js' ], // le librerie possono essere più di una
        function(errors, window) {
            callback(window);
        }
    );
}
```

Al wrapper **caricaDom** vengono passati i seguenti parametri :

- il path della pagina html da aggiornare
- il riferimento ad una funzione di callback scritta all'interno del chiamante e che **caricaDom** richiamerà dopo aver 'agganciato' il file html e la libreria jQuery, passandogli come parametro il puntatore al DOM della pagina web. Tramite questo puntatore il chiamante potrà accedere alla pagina ed eseguire tutte le modifiche necessarie.

```
caricaDom("myFile,html", function (window) {
    var $ = window.$;
    console.log($("#myTag").text());
});
```

Occorre prestare attenzione al fatto che **le modifiche apportate al dom html non vengono applicate direttamente alla variabile `page` restituita da readFileSync** (ammesso che essa sia visibile**) ma vengono applicate al DOM puntato da window**
Per cui se si vuole inviare al client la pagina modificata occorre utilizzare una delle seguenti sintassi:

```
res.end(page);   // Not Working
res.end(window.document.documentElement.outerHTML)
res.end( "<!doctype html>" + get..("html")[0].outerHTML;
res.end( "<!doctype html> <html>"  +  get..("html")[0].innerHTML + "</html>"
```

`docoumentElement` rappresenta il documento corrente (in pratica il tag html e tutto il suo contenuto)

Esempio di codice per l'aggiunta di un elenco di nomi ad in ListBox (sqLite)

```
caricaDom("myFile,html", function (window) {
   var $ = window.$;
    var db = new sqlite.Database(file);
    db.serialize(function() {
        db.each("SELECT nome FROM studenti",
        function(err, row) {   // per ogni riga :
            $("#lstStudenti").append("<option>"+row.nome+"</option>");
        },
        function(err, rows) {  // al termine :
            res.writeHead(200, { "Content-Type": "text/html" });
            res.end(window.document.documentElement.outerHTML)
        });
        db.close();
    });
});
```

htmlparser2

Molto migliorato rispetto alla versione 1. Riceve una stringa html e restituisce il DOM javascript classico.
La sintassi di utilizzo è pressappoco la seguente:

```
var handler = new htmlparser.DomHandler(function(err, dom) {
     // ... DO CODE HERE
})
new htmlparser.Parser(handler).parseComplete(html_string)
```

Il modulo net e la creazione di un Server TCP

Il modulo **net** permette di implementare applicazioni client-server sfruttando i socket come canale di comunicazione appoggiato direttamente su TCP. Nota:

- Il protocollo HTTP è un protocollo "stateless", nel senso che la connessione viene automaticamente chiusa dopo ogni scambio di informazioni.
- Il protocollo TCP è invece "statefull" perché, una volta stabilita la connessione, questa viene mantenuta aperta fino a quando uno dei due partner non provvede esplicitamente a chiuderla,

Gestione della connessione lato server

Il metodo **.createServer** consente di creare un generico server TCP in ascolto sulla porta indicata. Riceve come parametro una funzione di callback richiamata in corrispondenza di ogni **richiesta di connessione** da parte di un client. Alla funzione di callback viene iniettato come parametro un **socket object** contenente tutte le informazioni relative al socket client che ha richiesto la connessione.

```
net.createServer(function(socket){});
```

Gestione della connessione lato client

Il metodo **.connect**`(port[, host][, callback])` consente ad un TCP client di connettersi ad un TCP server. Se non si specifica l'host la richiesta viene inviata su localhost. Restituisce un oggetto **socket** di connessione al server che potrà essere utilizzato per inviare / ricevere dati dal server.

Proprietà, metodi ed eventi relativi all'oggetto socket

Una volta stabilita la connessione sia il client che il server possono inviare / ricevere dati sulla connessione stessa.

socket.write(str) consente sia al client che al server di scrivere stringhe sulla connessione.

- Se la stringa **str** passata a socket.write() è vuota, la write NON viene eseguita
- Più **write** consecutive possono essere automaticamente accorpate a discrezione del run time

socket.on("data", function(msg){}); evento richiamato ogni volta che il server o il client ricevono dei dati sul socket corrente. Se l'evento non viene gestito, i dati ricevuto andranno persi. Nell'es la funzione di gestione dell'evento riscrive nel socket il dato appena ricevuto, realizzando un sistema di eco: tutto ciò che il client scrive nel socket viene rimandato indietro al mittente.

socket.remoteAddress indirizzo IP dell'host remoto.
socket.remotePort porta dell'host remoto.

La chiusura della connessione

La chiusura della connessione deve essere richiesta dal client.

socket.end() consente al client di chiudere una connessione con il server.

socket.on("end", function(){}); evento generato sia sul server che sul client in corrispondenza della richiesta di chiusura della connessione da parte del client. Sul client viene generato subito dopo dell'invio della richiesta di chiusura.

Struttura di un server

```
var server = require("net") ;
server.createServer(rispondi).listen(1337) ;

// richiamata ad ogni richiesta di connessione
function rispondi(socket) {

    console.log("Richiesta da : " + socket.remotePort);
    socket.write("CONNESSIONE Accettata\r\n");

    // evento di ricezione nuovi dati dal client
    socket.on('data', function(msg){
        console.log(msg.toString());
        this.write(msg);
    });

    // evento di chiusura della connessione
    socket.on('end', function() {
         console.log("client " + socket.remotePort + " disconnesso");
    });
}
```

Scrittura di un client

```
var net = require('net');
var stdin = process.openStdin();

// L'indirizzo IP è opzionale. In sua assenza il default è 127.0.0.1
var client = net.connect(1337, '127.0.0.1', function() {
     console.log('Richiesta di connessione inviata al server ....');
});

client.on('data', function(data) {
     console.log(data.toString());
});

stdin.addListener("data", function(msg) {
    if (msg.toString()=="end\r\n")
        client.end();
    else
        client.write(msg);
});

client.on('end', function() {
     console.log("Disconnessione ok ...");
     process.exit();
});
```

Nota sulla scrittura di protocolli

Poiché la ricezione dei dati è gestita in modo asincrono (evento **on.data**), non è ovviamente possibile aprire una IF ed aspettare i dati successivi al suo interno come si fa con i socket in ANSI C.
L'unica possibile soluzione è quella di cambiare stato dopo il ricevimento di ogni singolo comando.

Esempio di chat

Supponendo di avere più utenti collegati in chat, il server dovrà tenersi un vettore di socket, uno per ogni utente. Il server ascolterà i messaggi ricevuti sulle varie connessioni, e li inoltrerà a tutti gli altri client.

```
var clients = [];
function rispondi(socket){
   console.log("Richiesta da : " + socket.remotePort);
   socket.write("CONNESSIONE OK, inserisci per prima cosa il nickname: ");
   var esistente=false;

   // ricezione dati dal client
   socket.on("data", function(data){
       var _client;
      // il primo messaggio ricevuto DEVE essere il nickname
      if(!esistente){
          var nick = data.toString();
          _client = {
             "nickname": nick,
             "ip": this.remoteAddress,
             "porta": this.remotePort,
             "toString":function(){
                return this.nickname + " - " + this.ip  + " : "+ this.porta;
             },
             "socket":this
          };
          console.log(_client.toString());
             clients.push(_client);
             this.write("Benvenuto " + _client.nickname + "\r\n");
             esistente=true;
          }
       else if(data.toString() == "LST"){
          for(var i=0; i<clients.length; i++){
                this.write(clients[i].toString() + "\r\n");
          }
       }
       else{
          _client = clients[cercaClient(this)];
          console.log(_client.nickname + ">> " + data.toString());
             for (var i=0; i<clients.length; i++)
                if(clients[i].socket != this)
          clients[i].socket.write(_client.nickname + ">> " + data.toString());
        }
   });

   socket.on("end",function(){
     console.log("client " + this.remotePort + " disconnesso");
     var pos = cercaClient(this);
     clients.splice(pos, 1);
   });
}

function cercaClient(_this){
   for (var i=0; i<clients.length; i++){
      if(clients[i].socket == _this) break;
   }
   return i;
}
```

Utilizzo di un client TELNET

Invece di scrivere un client TCP personalizzato, si può utilizzare un client telnet standard.
Telnet è un protocollo nato per il controllo di un terminale da remoto, in cui **ogni singolo carattere digitato** viene immediatamente inviato al server, generando ogni volta un nuovo evento .data

In ambiente Windows il cliente telnet non è abilitato di default ma occorre andare su:
Pannello di Controllo
/ Programmi e Funzionalità
/ Attivazione e Disattivazione delle funzionalità di Windows
Attivare la funzionalità **Client Telnet**

Aprire un terminale ed eseguire il client telnet nel modo seguente :

```
telnet localhost 1337     // oppure
telnet
   open localhost 1337
   ctrl +        consente di ritornare al prompt di Telnet, senza chiudere la connessione
   invio         consente di rientrare nella connessione
```

Dal prompt di telnet :

close consente di chiudere la connessione corrente (dopo essere ritornati al prompt con ctrl +)
quit esce da telnet ritornando al prompt del terminale.