



Node.js

Home Page del corso: <http://lia.disi.unibo.it/Courses/twt2021-info/>
Versione elettronica: 4.03.Node.js.pdf
Versione elettronica: 4.03.Node.js-2p.pdf

Verso nuovi modelli, più asincroni, più scalabili, ...

Verso nuovi modelli, non solo per applicazioni Web, più asincroni, più scalabili, ...

Esempio notevole: Node.js

- Idea centrale di supporto efficiente a I/O asincrono non-bloccante
- Tecnologia server-side
- NON utilizzo di thread/processi dedicati
- Maggiore scalabilità
- Verso esasperazione del concetto di «server stateless»

Node.js

- **Necessario server-side un runtime environment JavaScript che ospiti Google Chrome V8 engine**
 - Soluzione server-side per JavaScript
 - Codice JavaScript compilato, buona efficienza runtime
- **Progettato per estrema concorrenza e scalabilità**
 - Senza thread o processi dedicati, definiti ad esempio a livello di applicazione
- **SEMPRE NON bloccante, persino per chiamate I/O-oriented**

- **Event Loop**
 - Al posto di thread, usa un event loop con stack
 - Riduce fortemente overhead di context switching
- **Usa il framework CommonJS**
 - Leggermente più “simile” a un vero linguaggio di programmazione OO

Perché usare event loop e I/O asincrono?

Ordini di grandezza tipici al giorno d'oggi...

Operation	CPU Cycles
L1	3 cycles
L2	14 cycles
RAM	250 cycles
DISK	41 000 000 cycles
NETWORK	240 000 000 cycles

TABLE FROM RYAN DAHL'S 2009.11.08 PRESENTATION ON NODE.JS
[HTTPS://NODEJS.ORG/JSCONF.PDF](https://nodejs.org/jsconf.pdf)

Thread vs. Event-driven

Thread	Asynchronous Event-driven
Blocca applicazione/richieste con listener-worker thread	Un solo thread, che fa ripetutamente fetching di eventi da una coda
Usa modello incoming-request	Usa una coda di eventi e processa eventi presenti
Multithreaded server potrebbe bloccare una richiesta che coinvolge eventi multipli	Salva stato e passa poi a processare il prossimo evento in coda
Usa context switching	No contention e NO context switch
Usa ambienti multithreading in cui listener e worker thread spesso acquisiscono incoming-request lock	Usa framework con meccanismi per cosiddetto I/O asincrono (callback, NO poll/select, O_NONBLOCK)

Thread vs. Eventi

Threads

```
request = readRequest(socket);  
reply = processRequest(request);  
sendReply(socket, reply);
```

Implementation:

Thread switching (i.e. blocking) and a scheduler

versus

Events

```
startRequest(socket);  
listen("requestAvail", processRequest);  
listen("processDone", sendReplyToSock);
```

Implementation:

Event queue processing

Thread vs. Eventi usando Callback

```
request = readRequest(socket);  
reply = processRequest(request);  
sendReply(socket, reply);
```

Implementation:

Thread switching (i.e. blocking) and a scheduler

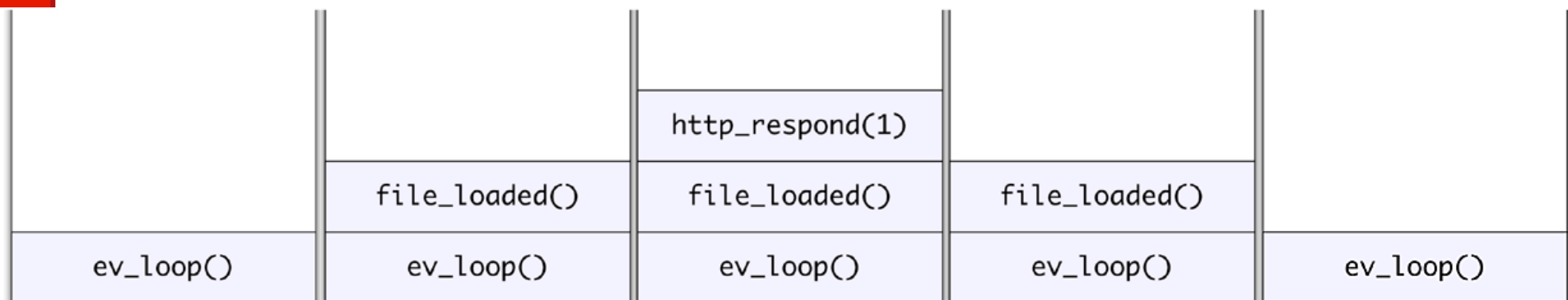
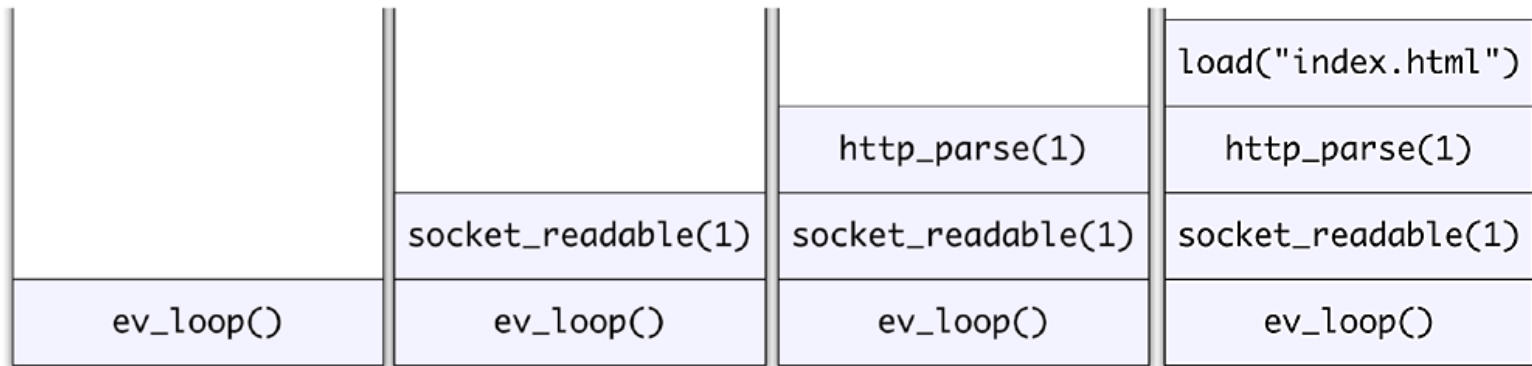
```
readRequest(socket, function(request) {  
    processRequest(request,  
        function (reply) {  
            sendReply(socket, reply);  
        });  
});
```

Implementation:

Event queue processing

Possibile esempio di Event Loop

- Arriva una richiesta per “index.html”
- Dopo aver processato primo stack di eventi, ev_loop è idle
- Caricamento di file da disco scatena nuovi eventi



Possibile esempio di Event Loop

- Come lavora event loop
- Esempio di lettura da socket

Inner loop

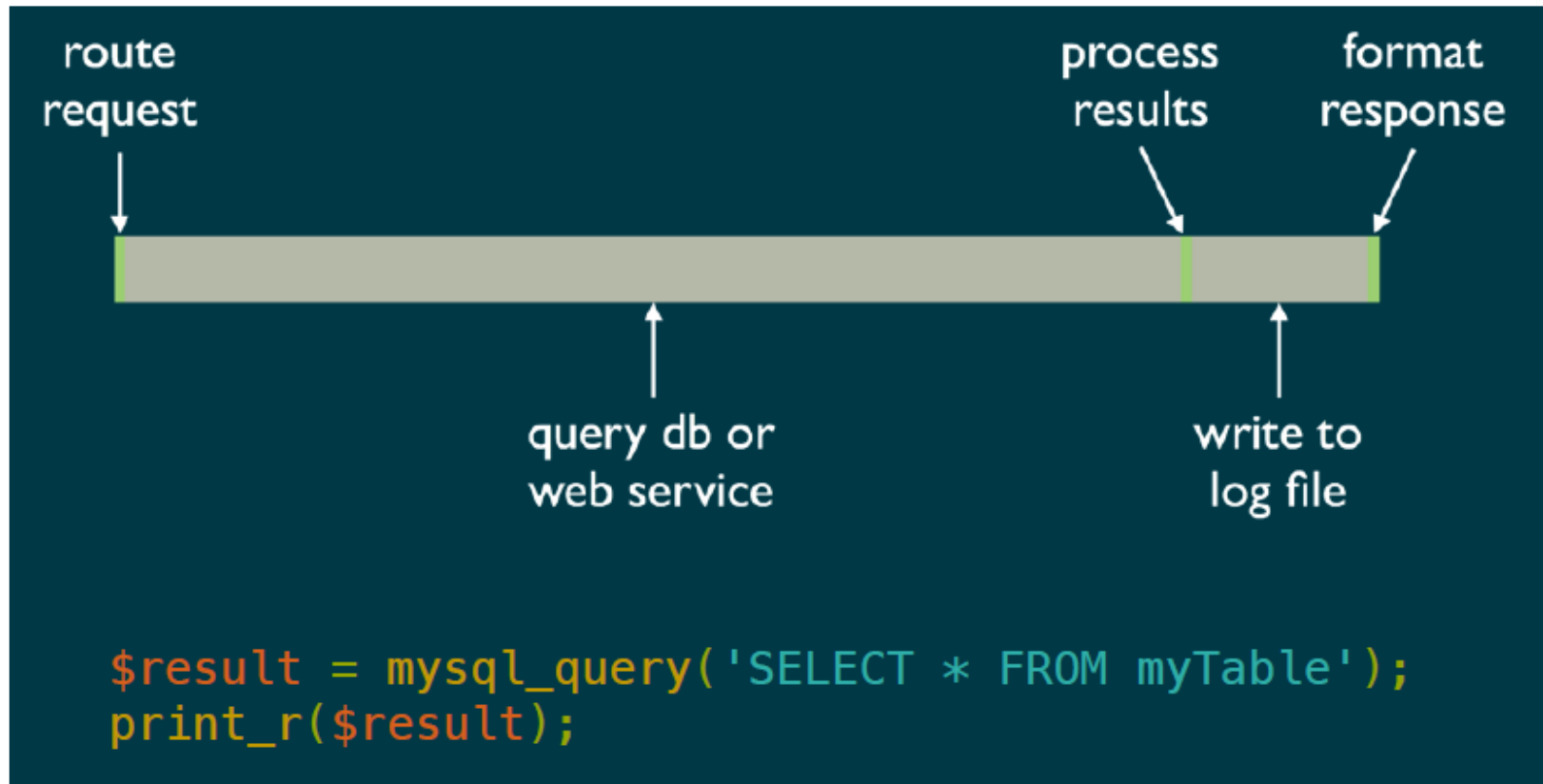
```
while (true) {  
    if (!eventQueue.isEmpty()) {  
        eventQueue.pop().call();  
    }  
}
```

Never wait/block in event handler . Example `readRequest(socket)`;

1. `launchReadRequest(socket); // Returns immediately`
2. When read finishes: `eventQueue.push(readDoneEventHandler);`

Operazioni I/O sincrone...

In molti linguaggi di prog e framework tradizionali, operazioni I/O sono bloccanti: bloccano il progresso di un thread in attesa di lettura da hard drive o da rete, ad esempio...



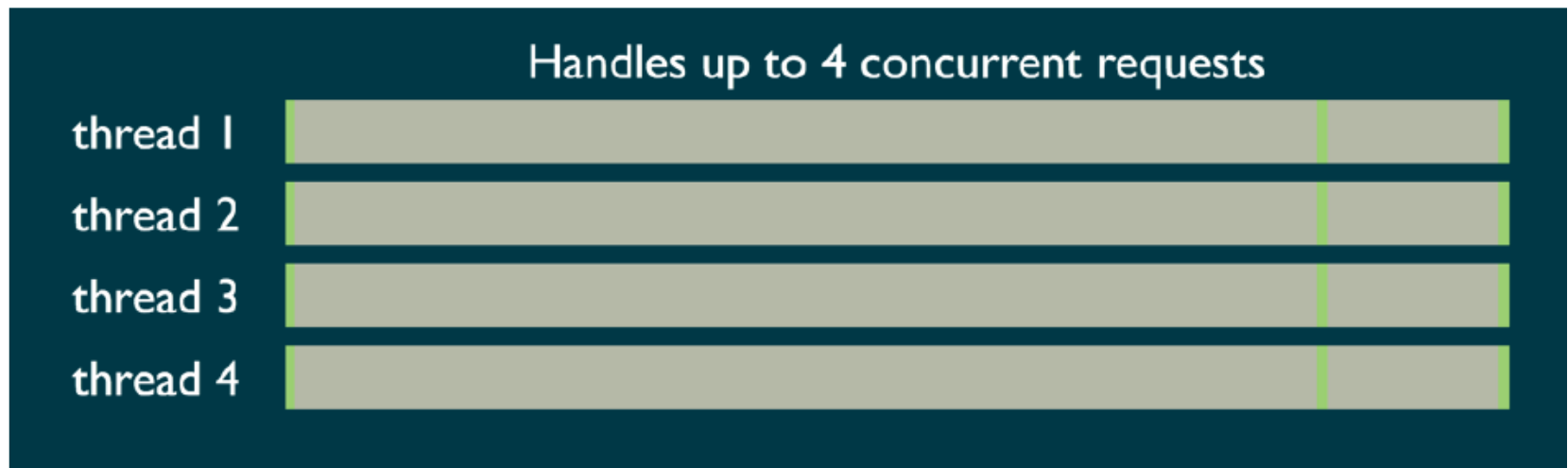
I/O bloccante e multi-threading

In caso di I/O bloccante, un server “tradizionale” usa multi-threading per limitare l’attesa

- Thread pool
- Un thread per connessione

Ma comunque ogni thread passa maggior parte del tempo in attesa di I/O

Andare verso altissimi numeri di thread introduce overhead di context switching e significativo uso di memoria



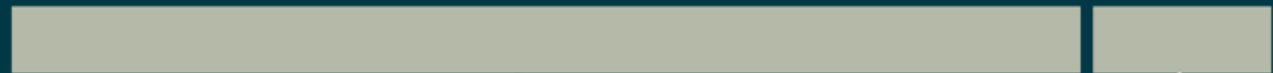
Node.js, single-thread e non-blocking I/O

Node.js usa un approccio single-thread non-blocking I/O: ogni funzione che fa operazioni I/O viene gestita in modo asincrono non-bloccante tramite callback

```
db.query("select..", function (result) {  
    doSomething(result);  
});  
nextTask();
```

Handles many concurrent requests in one process/thread

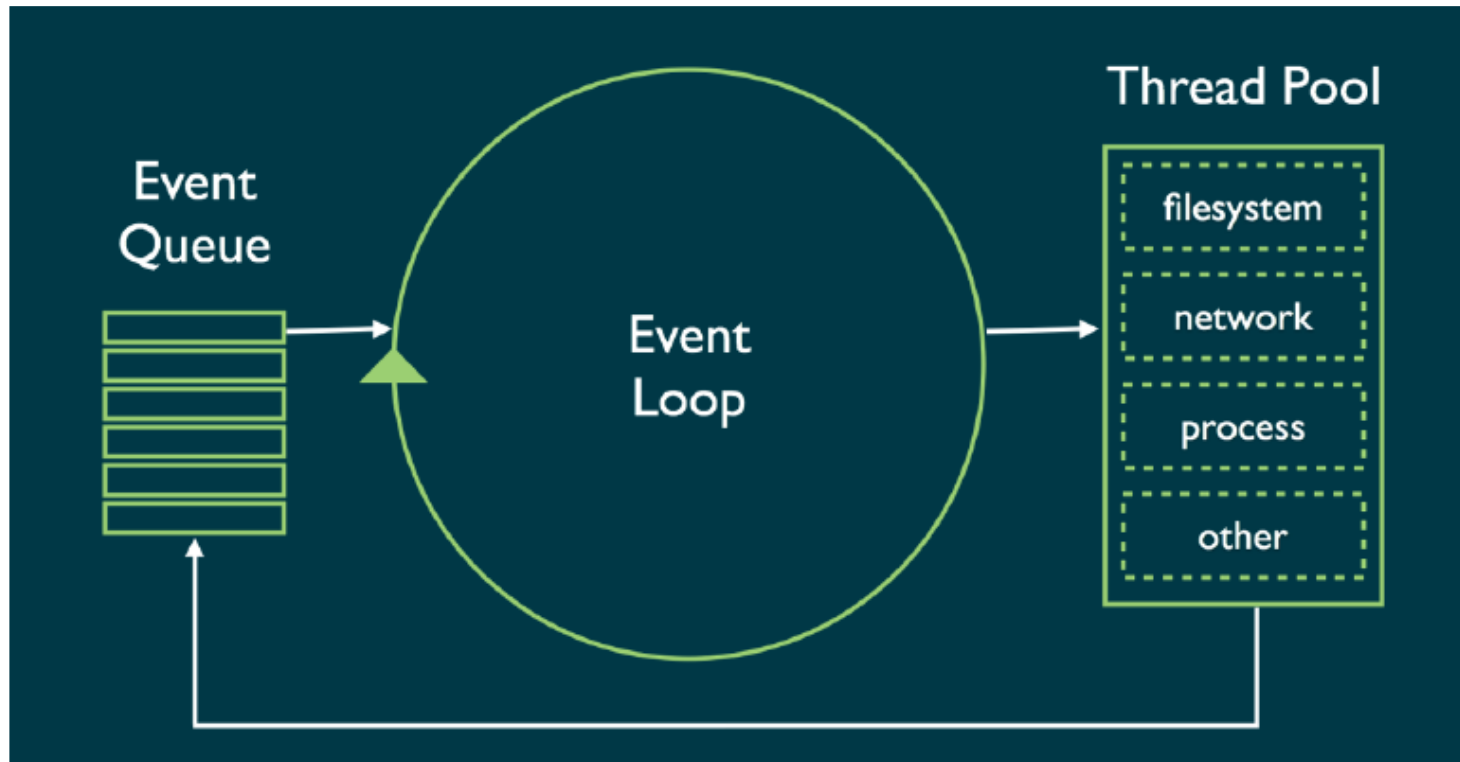
process 1



Thread Pool and Async I/O APIs

Attenzione a task CPU intensive!

- Utilizzando un thread singolo con un event loop, Node.js supporta decine di migliaia di connessioni concorrenti, senza costo di context switching
- **Task nell'event loop devono eseguire velocemente per NON bloccare la coda: attenzione a task CPU intensive**



Non-blocking I/O

- **Il servitore non compie nessuna altra operazione se non di I/O**
 - Infatti, applicazioni JavaScript (di scripting in generale) in attesa su richieste I/O tendono a degradare significativamente performance
- **Per evitare blocking, Node.js utilizza stessa natura event driven di JavaScript, associando callback alla ricezione di richieste I/O**
- **Script Node.js in attesa su I/O NON sprecano molte risorse (popped off dallo stack automaticamente quando il loro codice non-I/O related termina la sua esecuzione)**

Node.js in a nutshell

- **Utilizzo dello stesso JavaScript engine sia lato browser che server**
 - Ovviamente senza bisogno di DOM lato server
- **Gestione eventi su una coda degli eventi**
 - Ogni operazione esegue come una chiamata dall'event loop
- **Uso di interfaccia ad eventi per ogni operazione di SO**
 - Wrapping di tutte le chiamate bloccanti di SO (I/O su file e socket/network)
- **Uso di sistema di moduli (import/export)**
 - Moduli specializzati per il supporto a data management efficiente

Quando usare Node.js?

- **Node.js è particolarmente adatto alla creazione di Web server e strumenti di networking vari**
- **Uso di una collezione di moduli che realizzano varie funzionalità core:** per file system I/O, per networking, per funzioni crittografiche, per gestione stream di dati, ecc.
- **Insiemi di moduli (framework) per velocizzare lo sviluppo di applicazioni Web, come Express.js**

- **Non solo server-side (come già visto nell'esercitazione su React): diversi strumenti per sviluppo frontend e associate DevOps**

Esempio di uso di Node.js in server-side script PHP

Anche integrato in altri framework per il Web e non solo, ad esempio PHP

```
<?php
$result = mysql_query('SELECT * FROM ...');
while($r = mysql_fetch_array($result)){
    // Do something
}

// Wait for query processing to finish...
?>

<script type="text/javascript">
mysql.query('SELECT * FROM ...', function (err, result, fields){
    // Do something
});

// Don't wait, just continue executing
</script>
```

Stili di programmazione: thread vs callback

Threads

```
r1 = step1();
console.log('step1 done', r1);
r2 = step2(r1);
console.log('step2 done', r2);
r3 = step3(r2);
console.log('step3 done', r3);
console.log('All Done!');
```

Works for **non-blocking** calls in both styles

Callbacks

```
step1(function(r1) {
  console.log('step1 done', r1);
  step2(r1, function (r2) {
    console.log('step2 done', r2);
    step3(r2, function (r3) {
      console.log('step3 done', r3);
    });
  });
});
console.log('All Done!'); // Wrong!
```

Stili di programmazione: thread vs callback

Threads

```
r1 = step1();
console.log('step1 done', r1);
r2 = step2(r1);
console.log('step2 done', r2);
r3 = step3(r2);
console.log('step3 done', r3);
console.log('All Done!');
```

Callbacks

```
step1(function(r1) {
  console.log('step1 done', r1);
  step2(r1, function (r2) {
    console.log('step2 done', r2);
    step3(r2, function (r3) {
      console.log('step3 done', r3);
      console.log('All Done!');
    });
  });
});
```

Uniformità JavaScript lato cliente e server

Uso di JavaScript sia lato cliente che lato server

- No “context switch” di linguaggio di programmazione nella mente dei Web developer
- Ma ricordiamoci se stiamo programmando CLIENT O SERVER side...
 - Client-side JavaScript usa ampiamente DOM, tipicamente NO accesso a file o persistent storage
 - Server-side JavaScript lavora prevalentem con file e/o persistent storage, NO DOM

Moduli Node.js

Il core di Node consiste di circa una ventina di moduli, alcuni di più basso livello come per la gestione di eventi e stream, altri di più alto livello come http

```
// Carica il modulo http per creare un http server
var http=require('http');
// Configura HTTP server per rispondere con Hello World
var server=http.createServer(function(request,response) {
response.writeHead(200, {"Content-Type":"text/plain"});
response.end("Hello World\n");
});
// Ascolta su porta 8000
server.listen(8000);
// Scrive un messaggio sulla console terminale
console.log("Server running at http://127.0.0.1:8000/");
```

Moduli Node.js

- Il core di Node è stato progettato per essere piccolo e snello; i moduli che fanno parte del core si focalizzano su protocolli e formati di uso comune
- Per ogni altra cosa, si usa **npm**: chiunque può creare un modulo Node con funzionalità aggiuntive e pubblicarlo in npm

NPM

- Package manager di grande successo/crescita, che Semplifica sharing e riuso di codice JavaScript in forma di moduli
- Preinstallato con distribuzione Node
- Esegue tramite linea di comando e permette di ritrovare moduli dal registry pubblico in <http://npmjs.org>

Esempio di lettura di file in Node.js

```
var fs = require("fs"); // modulo fs richiesto
// oggetto fs fa da wrapper a chiamate bloccanti sui file
// read() a livello SO è sincrona bloccante mentre
// fs.readFile è non-bloccante
fs.readFile("smallFile", readDoneCallback); // Inizio lettura

function readDoneCallback(error, dataBuffer) {
// convenzione Node per callback: primo argomento è oggetto
// js di errore
if (!error) {
console.log("smallFile contents", dataBuffer.toString());
}
}
```


Listener/Emitter Pattern

- Ovviamente, **listener** come funzione da chiamare quando evento associato viene lanciato
- **Emitter** come segnale che un evento è accaduto
- L'emissione di un evento causa invocazione di **TUTTE** le funzioni listener

```
myEmitter.on('myEvent', function(param1, param2) {  
  console.log('myEvent occurred with ' + param1 + ' and '  
  + param2 + '!');  
});
```

```
myEmitter.emit('myEvent', 'arg1', 'arg2');
```

- In seguito a **emit**, i listener sono invocati in modo **sincrono-bloccante** e nell'ordine con cui sono stati registrati; **no listener, no op**

Node.js Stream (Flussi)

- Node contiene moduli che producono/consumano flussi di dati (stream)
- Può essere modo utile per strutturare server
 - Network interface \Leftrightarrow TCP/IP protocol processing \Leftrightarrow HTTP protocol processing \Leftrightarrow codice sviluppatore
- Si possono costruire stream anche dinamicamente e aggiungere moduli sul flusso
 - Ad esempio `stream.push(Encryption)`
 - Network interface \Leftrightarrow TCP/IP protocol processing \Leftrightarrow Encryption \Leftrightarrow HTTP processing
- Readable stream (es: `fs.createReadStream`)
- Writable stream (es. `fs.createWriteStream`)
- Duplex stream (es. `net.createConnection`)
- Transform stream (es. `zlib`, `crypto`)

Leggere File usando Stream

```
var readableStreamEvent =
    fs.createReadStream("bigFile");
readableStreamEvent.on('data', function (chunkBuffer) {
    console.log('got chunk of', chunkBuffer.length,
        'bytes');
});

readableStreamEvent.on('end', function() {
// Lanciato dopo che sono stati letti tutti i datachunk
console.log('got all the data');
});

readableStreamEvent.on('error', function (err) {
console.error('got error', err);
});
```

Scrivere File usando Stream

```
var writableStreamEvent =  
    fs.createWriteStream('outputFile');  
  
writableStreamEvent.on('finish', function () {  
    console.log('file has been written!');  
});  
  
writableStreamEvent.write('Hello world!\n');  
writableStreamEvent.end();
```

TCP Networking in Node.js

- Esiste un modulo di rete Node, chiamato `net`, che fa da wrapper per le chiamate di rete di SO
- Include anche funzionalità di alto livello, come:

```
var net = require('net');  
net.createServer(processTCPconnection).  
  listen(4000);
```

- Crea una socket, fa binding su porta 4000 e si mette in stato di `listen` per connessioni
- Per ogni connessione TCP, invoca la funzione `processTCPconnection`

Esempio di Server per Chat

```
var clients = []; // Lista di client connessi
function processTCPconnection(socket) {
clients.push(socket); // Aggiunge il cliente alla lista
socket.on('data', function (data) {
    broadcast( "> " + data, socket);
    // invia a tutti i dati ricevuti });
socket.on('end', function () {
clients.splice(clients.indexOf(socket), 1); // remove
socket
}); }
// invia messaggio a tutti i clienti
function broadcast(message, sender) {
clients.forEach(function (client) {
if (client === sender) return;
client.write(message); });
}
```

Modulo Express

- **Express.js è il framework più utilizzato oggi per lo sviluppo di applicazioni Web su Node**
- **Ispirato e basato sul precedente Sinatra**

Principali caratteristiche:

- **Focus su high performance**
- **Diverse opzioni e motori di templating**
- **Eseguibili per rapida generazione di applicazioni**

```
var express=require('express');
var app=express();
app.get('/',function(req,res) {
    res.send('Hello World!'); });
var server=app.listen(3000,function() {
    var host=server.address().address;
    var port=server.address().port;
    console.log('Listening at http://%s:%s',host,port); });
```

Lista di Moduli Disponibili per Node.js

Core Module	Description
<u>http</u>	http module includes classes, methods and events to create Node.js http server.
<u>url</u>	url module includes methods for URL resolution and parsing.
<u>querystring</u>	querystring module includes methods to deal with query string.
<u>path</u>	path module includes methods to deal with file paths.
<u>fs</u>	fs module includes classes, methods, and events to work with file I/O.
<u>util</u>	util module includes utility functions useful for programmers.

Una lista abbastanza completa dei moduli Node.js di maggiore utilizzo è reperibile qui:

https://www.w3schools.com/nodejs/ref_modules.asp

Verso nuovi modelli, più asincroni, più scalabili, ...

Esempio notevole: Node.js

- Idea centrale di supporto efficiente a I/O asincrono non-bloccante
- Tecnologia server-side
- NON utilizzo di thread/processi dedicati
- Maggiore scalabilità
- Verso esasperazione del concetto di «server stateless»

Alcuni possibili riferimenti

- <http://nodejs.org/>
- <https://howtonode.org/>
- M. Wandschneider, “Node.js: Creare applicazioni web in JavaScript”, Apogeo, 2013
- M. Cantelon, M. Harter, T.J. Holowaychuk, “Node.js in Action”, Manning, 2013
- <http://ajaxian.com/archives/google-chrome-chromium-and-v8>
- <http://news.softpedia.com/news/IE9-RC-vs-Chrome-10-9-vs-Opera-11-vs-Firefox-11-Performance-Comparison-183973.shtml>