

Appunti su Node.js, npm, Express.js e MongoDB (23-Javascript-6-Node.txt)

Javascript Server-Side: Node.js e l'Ecosistema

- **Stack MEAN:** Un insieme di tecnologie JavaScript per lo sviluppo web full-stack:
 - **MongoDB:** Database NoSQL per la persistenza dei dati.
 - **Express.js:** Framework per applicazioni web lato server.
 - **Angular.js:** Framework MVC per la creazione di template HTML dinamici (lato client).
 - **Node.js:** Piattaforma software per applicazioni server-side.
 - Alternativa allo stack LAMP (Linux, Apache, MySQL, PHP/Python/Perl).
- **Node.js:**
 - Ambiente di esecuzione JavaScript server-side.
 - Progettato per applicazioni efficienti.
 - Basato sul motore JavaScript V8 (lo stesso di Google Chrome).
 - Vasto ecosistema di moduli gestito da `npm` (Node Package Manager).
 - Modello di governance aperto (Node.js Foundation).
- **Esecuzione Single-Thread Non-Blocking:**
 - Node.js utilizza un singolo thread (event loop) per gestire tutte le richieste.
 - Maggiore efficienza rispetto ai modelli multi-thread tradizionali (meno overhead).
 - Funzioni asincrone e callback per operazioni I/O non bloccanti.
 - Fondamentale evitare operazioni onerose nel main loop (rischio di bloccare l'applicazione).
- **Node.js come Motore HTTP:**
 - Può sostituire soluzioni CGI tradizionali (PHP, Python, Perl, ecc.).
 - Esempio: semplice server che restituisce "Hello World" in HTML.

```
var http = require('http');
http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/html'});
  response.end('<h1>Hello World</h1>');
}).listen(8000);
```

- **Node.js con Altri Protocolli:**
 - Può implementare protocolli diversi da HTTP.
 - Esempio: semplice server TCP.

Event Loop

- Elemento centrale del motore Node.js.
- Entra in funzione all'avvio dello script ed esce quando non ci sono più callback pendenti.
- Utilizza la libreria `libuv` per operazioni I/O asincrone (con un pool di thread interno).
- Comunicazione tramite emissione di eventi (Event Emitter).
 - `eventEmitter.on(eventName, listener)` : associa una funzione (listener) a un evento.

- Le funzioni associate vengono invocate in modo sincrono al verificarsi dell'evento.

Asincronia e Funzioni Callback

- Node.js è basato quasi esclusivamente su funzioni asincrone e callback.
- **Convenzione error-first:**
 - Le funzioni callback accettano come primo parametro un eventuale errore.
 - Esempio:

```
fs.readFile('test.txt', function(err, data) {  
  if(err) {  
    console.log('Error: '+error.message);  
    return;  
  }  
  doSomething(data);  
});
```

Moduli Node.js

- Permettono di includere altri file JavaScript e riutilizzare librerie.
- Favoriscono l'organizzazione del codice in parti indipendenti.
- **Sistema di caricamento dei moduli:**
 - Un modulo è un file JavaScript.
 - `require(<modulo>)` : include un modulo.
 - Ricerca del modulo: locale (`./`), globale (`NODE_PATH`), core (built-in).
- **Tipi di moduli:**
 - **Core:** Integrati nel sistema, non necessitano di installazione.
 - **Dipendenze locali:** Installate per l'applicazione corrente nella directory `./node_modules/`.
 - **Dipendenze globali:** Disponibili per tutte le applicazioni (directory specificate in `NODE_PATH`).
- **Creazione di un modulo:**
 - Esportazione di funzioni o oggetti tramite `module.exports` . *Esempio CommonJS

```
// greetings.js  
hello = function() {  
  console.log("\n Hello! \n")  
}  
ciao = function() {  
  console.log("\n Ciao! \n")  
}  
module.exports.hello = hello;  
module.exports.ciao = ciao;
```

*Esempio con sintassi ES Modules (ESM)

```
// greetings.js
hello = function() {
  return("\n Hello! \n")
}
ciao = function() {
  return("\n Ciao! \n")
}
greetings = {hello, ciao}
export default greetings;
```

npm (Node Package Manager)

- Gestisce la distribuzione e l'installazione dei moduli Node.js.
- Interagisce con il registro npm.
- Gestione delle dipendenze e delle versioni dei pacchetti.
- Semplice processo di pubblicazione dei pacchetti.
- Piattaforma per repository pubblici e privati, servizi enterprise, ecc.
- **Creazione di un pacchetto:**
 - `npm init` : crea il file `package.json` (manifest) tramite un'interfaccia interattiva.
 - `package.json` : contiene metadati del pacchetto (nome, autore, dipendenze, ecc.).
- **Installazione di un pacchetto:**
 - `npm install <pacchetto>` : installa un pacchetto e le sue dipendenze nella directory `./node_modules/` .(Aggiorna `package.json`)
 - `npm install <pacchetto> --global` o `-g` : installa globalmente.
- **Comandi npm utili:**
 - `npm list` , `npm config list` , `npm search` , `npm install` , `npm uninstall` , `npm update` , `npm init` .

Express.js

- Framework per applicazioni web server-side per Node.js.
- Open-source (licenza MIT).
- Semplice, espandibile con plugin.
- Gestisce routing, sessioni, autenticazione, ecc.
- **Basic server:**

```
express = require("express")
app = express()
docs_handler= function(request, response){
  var docs = {server : "express"}
  response.json(docs);
}
```

```

}
app.get("/docs", docs_handler);
app.listen(8099, function(){
  console.log("\nExpressis running!!! \n")
})

```

- `require("express")` : include il modulo Express.
- `express()` : crea un'istanza dell'applicazione.
- `app.get(path, handler)` : definisce un handler per le richieste GET a un determinato percorso.
- `response.json(data)` : invia una risposta in formato JSON.
- `app.listen(port, callback)` : avvia il server su una porta specifica.

- **Routing:**

- `app.method(path, handler)` : definisce un handler per un metodo HTTP specifico (GET, POST, PUT, DELETE, ecc.).
- `path` : percorso dell'URI.
- `handler(request, response)` : funzione che gestisce la richiesta.
- Oggetti `request` e `response` : contengono informazioni sulla richiesta e sulla risposta HTTP.

- **Route paths:**

- Stringhe o espressioni regolari.
- Esempi: `*ab*`, `abc?de`, `abc+`, `store$`.

- **Routing parametrico:**

- `app.get('/users/:userId/sales/:saleId', ...)` : parametri nell'URI (`req.params`).
- Delegare la gestione del routing a moduli indipendenti con `express.Router()`.

```

// save as informatica.js
const express = require('express')
const router = express.Router()

router.get('/', (req, res) => {
  res.send('Home page dipartimentoInformatica')
})
router.get('/about', (req, res) => {
  res.send('A proposito del dipartimento')
})
module.exports = router

const info = require('./informatica.js')
// ...
app.use('/cs/', info)} //questo modulo risponde a URI del
tipo/cs/e/cs/about

```

- **Esempio routing REST:**

```
app.get("/clients/", getListaClienti);
app.post("/clients/", aggiungiCliente);
app.put("/clients/:id", creaModificaCliente);
app.get("/clients/:id", getCliente);
app.delete("/clients/:id", cancellaCliente);
```

- **Oggetti Request e Response:**

- Metodi per accedere a tutte le informazioni di richieste e risposte HTTP.
- Request: `body` , `query` , `params` , `cookies` , `headers` , ecc.
- Response: metodi per inviare status code, risposte in diversi formati, aggiungere header.

- **Accesso ai dati di un POST:**

- Libreria `body-parser` (integrata con Express):
 - `request.body.<post_variable>` : accesso ai dati inviati tramite POST.

- **Spedizione di una risposta:**

- Diversi metodi, es `res.json()` , `res.send()` , `res.sendFile()`

- **Middleware in Express:**

- Express ha poche funzionalità proprie (routing).
- Utilizza librerie middleware personalizzabili (stack di servizi).
- `app.use(<middleware>)` : aggiunge un middleware allo stack.
- Un'applicazione Express è una sequenza di chiamate a funzioni middleware.
- Il middleware può:
 - Accedere e modificare gli oggetti di richiesta e risposta.
 - Eseguire codice.
 - Chiamare il prossimo middleware (`next()`).
 - Terminare il ciclo e inviare la risposta.

- **File statici:**

- `express.static(directory)` : middleware per servire file statici (immagini, CSS, ecc.).
- `app.use("/images", express.static('images'))` : associa un path virtuale a una directory fisica.

- **Routing Modulare**

```
```Javascript
var express = require('express')
var router = express.Router()

router.get('/', function (req, res) {
 res.send("TODO. Elenco utenti")
})
router.get('/:name/', function (req, res) {
 res.send("TODO. Utente " + req.params.name)
})

module.exports = router
```

```
usersRouter = require('./usersRouter.js');
app.use("/users/", usersRouter);
...
```

- **Express.js e autenticazione:**

- Realizzata tramite middleware (es. Passport.js).
- Passport.js: flessibile, modulare, supporta diverse strategie di autenticazione.
- JWT (JSON Web Token): `express-jwt` .

- **Express.js e CORS:**

- Middleware `cors` : aggiunge gli header `Origin` e `Access-Control-Allow-Origin` .
- Abilita richieste Ajax cross-domain.

```
cors = require('cors');
app.use(cors())
app.options('*', cors())
```

- **Handlebar.js**

- Linguaggio di template "logicless"

- **Passport.js**

- Autenticazione con dozzine di package aggiuntivi

- **nodemon**

- Riavvia automaticamente node.js al cambiamento dei file

## MongoDB e Mongoose

- **MongoDB:**

- Database NoSQL orientato ai documenti.
- Documenti in formato JSON.
- Collezioni (eterogenee, senza schema fisso).
- Nessuna relazione tra collezioni garantita da MongoDB.
- Driver JavaScript lato server per Node.js.

- **Concetti chiave:**

- **Collezioni:** Struttura fondamentale (simile a tabelle relazionali o directory).
- **Documenti:** Strutture contenenti dati omogenei (simili a righe di una tabella).
- **Campi (proprietà, attributi):** Contengono valori semplici o riferimenti (simili a celle).
- **Schemi (con Mongoose):** Possibilità di definire schemi e forzare l'omogeneità.
- **Schema type:** Tipi di dati per i campi (String, Number, Boolean, ecc.).
- **Modelli:** Costruttori di alto livello che generano istanze di documenti da uno schema.

- **Operazioni CRUD (Create, Read, Update, Delete):**

- Creazione implicita di database e collezioni.
- `insert(data)` / `insertMany(data)` : inserimento di documenti.
- `find(query)` : interrogazione (con operatori come `$gt` , `$or` , ecc.).
- `updateOne` , `updateMany` , `replaceOne` : aggiornamento.

- `deleteOne` , `deleteMany` : eliminazione.

- **Connessione al server MongoDB (esempio):**

```
var client = require('mongodb').MongoClient;
client.connect("mongodb://site2223XX:[PWD]@mongosite2223XX?
writeConcern=majority",
 async function(error, db) {
 if(!error) {
 var people= db.collection("people");
 await people.insert({ nome: "Fabio", cognome: "Vitali"});
 db.close();
 }
 });
```

- **Mongoose:**

- Permette di imporre schemi sui documenti MongoDB.
- Definizione di schemi e modelli.
- Esempio:

```
// people.js
let mongoose = require('mongoose');
let people = new mongoose.Schema({
 name: String,
 surname: String,
 email: String,
 age: Number
});
module.exports= mongoose.model('Person', people);
```

```
let Person = require('./people.js');

let fv= new Person({
 name: "Fabio",
 surname: "Vitali",
 email: "fabio.vitali@unibo.it",
 age: 28
});

fv.save()
 .then((doc) => {console.log(doc);})
 .catch((err) => {console.error(err);});
```

- **MongoDB e il progetto (Gocker):**

- Utilizzare Gocker per lanciare MongoDB in un container separato ( `create mongoDBsite2223XX` ).
- Conservare username, password e hostname forniti da Gocker.
- MongoDB è accessibile solo all'interno dell'installazione Gocker.