



## React con Node.js ed Express

24 Gen,2021   lucio   blog, express,nodeJs, react   185 views

Tempo di lettura: 12 minuti

Nella [guida](#) relativa a React Hooks abbiamo realizzato una piccola applicazione chiamata Heroes. In questo tutorial la riprenderemo e vedremo come integrare React con Node.js ed Express.

Se siete curiosi di sapere come abbiamo sviluppato Heroes potete leggervi il [tutorial](#) , altrimenti scaricatevi l'applicazione da [Git Hub](#).

Mi aspetto che abbiate quanto meno le basi della libreria e di ESx, fondamentali per poter seguire la guida, altrimenti potete accedere al mio corso su [Udemy](#)(Chiedetemi pure un coupon).

## React con Node.js ed Express: Introduzione

Come dicevo in precedenza se non avete seguito i tutorial precedenti iniziate con il clonarvi il Repository relativo ad Heroes da [Git Hub](#), vi ricordo che per far ciò dovete installare [Git bash](#) nel vostro pc.

Scegliete una directory a piacere , fate click-sx con il mouse quindi scegliete [git Bash Here](#) , vi si aprirà git Bash nella posizione relativa la directory da voi scelta. Ora scrivete:

```
git clone https://github.com/freewebsolution/React2.0.git heroes
```

heroes è il nome che sceglieremo per l'App.

Perfetto ora spostatevi all'interno di essa:

```
cd heroes/
```

quindi installate la directory node\_modules, è chiaro che sul vostro pc deve essere installato [Node.js](#):

```
npm install
```

testiamola con

```
npm start
```

dovrebbe avviarsi senza problemi:

```
heroes@0.1.0 start F:\APP\REACT\forBlog\heroes
> concurrently --kill-others "npm run start-react" "npm run json-server"
[1]
[1] > heroes@0.1.0 json-server F:\APP\REACT\forBlog\heroes
[1] > json-server -p 3003 --watch db.json
[1]
[0]
[0] > heroes@0.1.0 start-react F:\APP\REACT\forBlog\heroes
[0] > react-scripts start
[0]
[1]
[1] \{\^_\^}/ hi!
[1]
[1] Loading db.json
[1] Done
```

```
[1]
[1] Resources
[1] http://localhost:3003/heroes
[1]
[1] Home
[1] http://localhost:3003
[1]
[1] Type s + enter at any time to create a snapshot of the database
[1] Watching...
[1]
[0] Starting the development server...
[0]
[0] Browserslist: caniuse-lite is outdated. Please run next command `npm update`
[0] Compiled successfully!
[0]
[0] You can now view heroes in the browser.
[0]
[0] Local: http://localhost:3000/
[0] On Your Network: http://192.168.10.1:3000/
[0]
```

come potete vedere ho fatto in modo che si avviasse l'applicazione e json-server alla porta **3003**.

Per il momento chiudete la connessione con il comando **ctrl+c** e concentriamoci sul backend dell'applicazione.

## React con Node.js ed Express: il Back end

Realizzeremo il back end su **Node.js** che è una runtime di Js basato su Google's **Chrome V8** JavaScript engine.

Realizzate una nuova directory e chiamatela backend, quindi come fatto prima apritela con gitBash (click-dx -> **git Bash Here**) ed inizializzate **il file package.json**:

```
npm init
```

rispondete alle domande dando invio, per evitare le domande potevamo flaggare in questo modo:

```
npm init -y
```

aprite la directory con il vostro ide, vi consiglio di utilizzare **Visual Studio Code**, editor gratuito non che perfetto per il nostro scopo.

Tricks: se siete su windows, da terminale digitate **code** . e vi si apre Visual Studio Code relativamente alla directory in cui vi trovate da linea di comando. Se siete su Linux o Mac leggetevi questa [guida](#).

Potete notare il file **package.json** :

```
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Lucio Ticali",
  "license": "ISC"
}
```

Il file definisce , ad esempio, che il punto di ingresso è **index.js** . Facciamo una piccola modifica all'oggetto **scripts** :

```
...
  "scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
...
```

ho aggiunto il nodo **start** con valore **node index.js** .

Nella radice del progetto createvi il file **index.js** quindi scrivete:

```
console.log('Hello world')
```

da ora in poi possiamo utilizzare il terminale di Visual Studio Code, per aprirlo digitate **ctrl+ò** , quindi una volta aperto eseguite il programma con Node:

```
node index.js
```

in alternativa possiamo eseguirlo digitando :

```
npm start
```

visto che lo abbiamo definito in **package.json**:

```
...  
  "scripts": {  
    "start": "node index.js",  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  ...
```

ad ogni modo sul terminale vedrete:

```
> backend@1.0.0 start F:\APP\REACT\forBlog\backend  
> node index.js  
Hello world  
lucio@LENOVO MINGW64 /f/APP/REACT/forBlog/backend
```

## Realizza un semplice Server web

Aprirete il file **index.js** e modificate come di seguito:

```
const http = require('http')  
const app = http.createServer((request, response) => {  
  response.writeHead(200, { 'Content-Type': 'text/plain' })  
  response.end('Hello World')  
})  
const PORT = 3001  
app.listen(PORT)  
console.log(`Server running on port ${PORT}`)
```

Una volta in esecuzione nella console verrà stampato il seguente messaggio:

```
lucio@LENOVO MINGW64 /f/APP/REACT/forBlog/backend  
$ npm start  
> backend@1.0.0 start F:\APP\REACT\forBlog\backend  
> node index.js  
Server running on port 3001
```

Nel browser aprite l'applicazione all'indirizzo: <http://localhost:3001>:



Anche se aggiungessimo del contenuto alla fine dell'Url il server continuerebbe a funzionare correttamente continuando a mostrare il contenuto della pagina.

NB: se la porta 3001 fosse utilizzata da un'altra applicazione l'avvio del server comporterebbe un errore del genere:

```
→ hello npm start
> hello@1.0.0 start /Users/mluukkai/opetus/_2019fullstack-code/part3/hello
> node index.js
Server running on port 3001
events.js:167
      throw er; // Unhandled 'error' event
      ^
Error: listen EADDRINUSE :::3001
    at Server.setupListenHandle [as _listen2] (net.js:1330:14)
    at listenInCluster (net.js:1378:12)
```

quindi in tal caso o chiudi una delle due applicazioni oppure cambi la porta, **index.js**:

```
const http = require('http')
const app = http.createServer((request, response) => {
  response.writeHead(200, { 'Content-Type': 'text/plain' })
  response.end('Hello World')
})
const PORT = 3001
app.listen(PORT)
console.log(`Server running on port ${PORT}`)
```

ad esempio:

```
const http = require('http')
const app = http.createServer((request, response) => {
  response.writeHead(200, { 'Content-Type': 'text/plain' })
  response.end('Hello World')
})
const PORT = 3002
app.listen(PORT)
console.log(`Server running on port ${PORT}`)
```

abbiamo cambiato la porta da 3001 a 3002. Vi ricordo che per terminare l'applicazione dovete digitare **ctrl+c** , quindi riavviate la:

```
lucio@LENOVO MINGW64 /f/APP/REACT/forBlog/backend
$ npm start
> backend@1.0.0 start F:\APP\REACT\forBlog\backend
> node index.js
Server running on port 3002
```

Analizziamo il codice:

```
const http = require('http')
```

Nella prima riga l'applicazione importa il modulo **web-server** integrato in Node. nella parte successiva:

```
const app = http.createServer((request, response) => {
  response.writeHead(200, { 'Content-Type': 'text/plain' })
  response.end('Hello World')
})
```

Il codice utilizza il metodo **createserver** del modulo **http** per creare un nuovo server web. Un gestore di eventi viene registrato sul server e viene chiamato ogni volta che si effettua una richiesta HTTP all'indirizzo **http://localhost:3001**.

La richiesta riceve una risposta di codice stato 200 con l'intestazione Content-Type impostata su text / plain e il contenuto del sito da restituire impostato su Hello World .

Le ultime righe collegano il server http assegnato alla variabile **app** , per ascoltare le richieste HTTP inviate alla porta 3002:

```
const PORT = 3002
app.listen(PORT)
console.log(`Server running on port ${PORT}`)
```

Lo scopo del server è quello di restituire i dati json al Frontend. Quindi modifichiamo il file **index.js** in modo tale che possa restituire l'elenco degli eroi hardcoded in formato JSON:

```
const http = require('http')
let heroes = [
  {
    "id": "hero_j1m8m4b",
    "name": "Captain America"
  },
  {
    "id": "hero_j1m8oq0c",
    "name": "Iron Man"
  },
  {
    "id": "hero_j1m8p3fd",
    "name": "Hulk"
  }
]
const app = http.createServer((request, response) => {
  response.writeHead(200, { 'Content-Type': 'text/plain' })
  response.end(JSON.stringify(heroes))
})
const PORT = 3002
app.listen(PORT)
console.log(`Server running on port ${PORT}`)
```

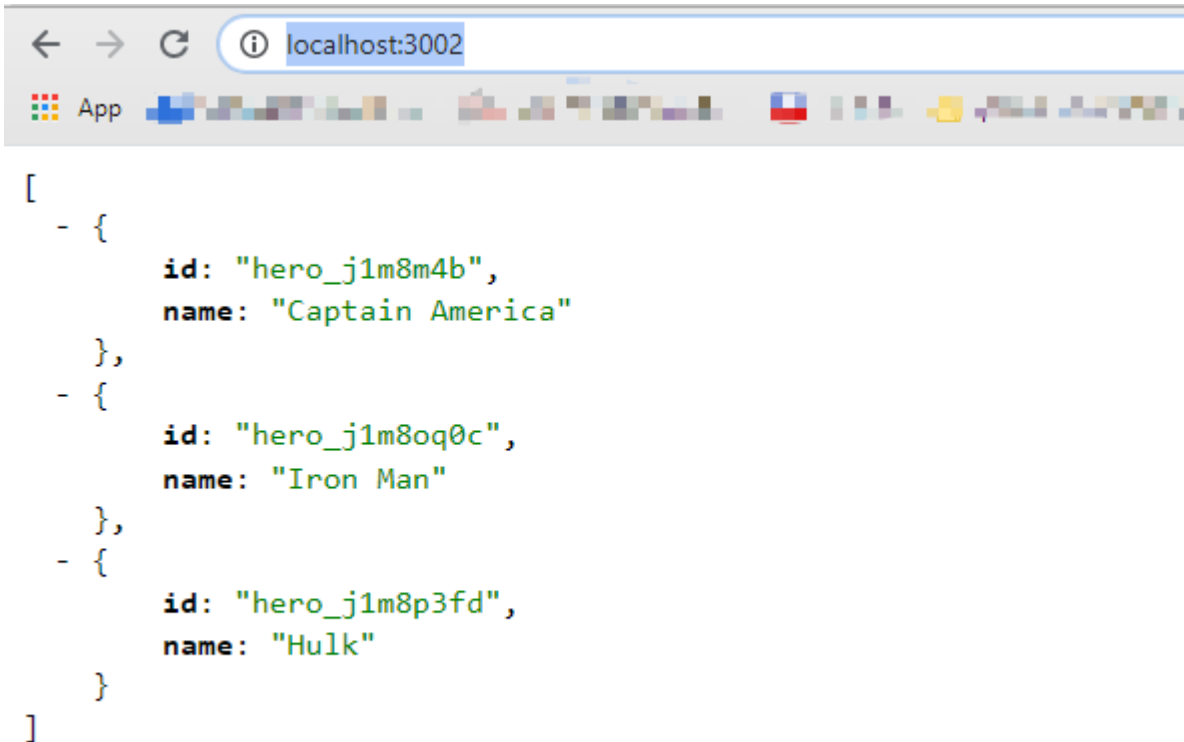
quindi avviate il server:

```
npm start
```

Il valore **application/json** nell'intestazione **Content-type** informa il destinatario che si tratta di dati JSON. L'array **heroes** viene trasformato in JSON mediante il metodo **JSON.stringify(heroes)**.

Ora aprite il browser all'indirizzo <http://localhost:3002/> ed apprezzate il risultato:





```
[
  - {
    id: "hero_j1m8m4b",
    name: "Captain America"
  },
  - {
    id: "hero_j1m8oq0c",
    name: "Iron Man"
  },
  - {
    id: "hero_j1m8p3fd",
    name: "Hulk"
  }
]
```

## Express

Anche se è possibile implementare tutto il codice relativo alla realizzazione del server con il **server web http** integrato in Node, non è consigliabile soprattutto se questa cresce.

Quindi è opportuno utilizzare una delle tante librerie che mirano a questo scopo, in poche parole sono molto più semplici da implementare. Una delle più note è sicuramente **Express**.

Installiamolo come dipendenza, dalla root principale digitiamo:

```
npm install express
```

questa viene aggiunta anche sul file **package.json**:

```
...
"dependencies": {
```

```
"express": "^4.17.1"
}
```

se vi spostate all'interno della directory **node\_modules** :

```
cd node_modules
```

quindi digitate:

```
ls
```

troverete tutte le dipendenze installate di express, queste sono dette dipendenze transitive del nostro progetto:

```
$ cd node_modules/
lucio@LENOVO MINGW64 /f/APP/REACT/forBlog/backend/node_modules
$ ls
accepts          cookie-signature  etag              inherits          mime-types        qs
setprototypeof   debug             express           ipaddr.js         ms                 range-
parser statuses   body-parser       depd              finalhandler      media-typer       negotiator        raw-body
toidentifier     bytes            destroy           forwarded          merge-descriptors on-finished       safe-
buffer type-is   content-disposition ee-first          fresh             methods           parseurl          safer-
buffer unpipe   content-type       encodeurl         http-errors       mime               path-to-regexp    send
utils-merge      cookie           escape-html       iconv-lite        mime-db            proxy-addr        serve-
static vary
lucio@LENOVO MINGW64 /f/APP/REACT/forBlog/backend/node_modules
```

La versione 4.17.1. di express è stato installata nel nostro progetto. Cosa significa il cursore davanti al numero di versione in package.json ?

```
"express": "^4.17.1"
```

Il modello di controllo delle versioni utilizzato in npm è chiamato controllo **delle versioni semantico** .

Il cursore davanti a `^ 4.17.1` significa che se e quando le dipendenze di un progetto vengono aggiornate, la versione di express installata sarà almeno la 4.17.1 . Tuttavia, la versione installata di express può anche essere quella che ha un numero di patch maggiore (l'ultimo numero) o un numero minore maggiore (il numero centrale). La versione principale della libreria indicata dal primo numero principale deve essere la stessa.

Possiamo aggiornare le dipendenze del progetto con il comando:

```
npm update
```

Allo stesso modo, se iniziamo a lavorare sul progetto su un altro computer, possiamo installare tutte le dipendenze aggiornate del progetto definite in package.json con il comando:

```
npm install
```

Se il numero maggiore di una dipendenza non cambia, le versioni più recenti dovrebbero essere **compatibili con le versioni precedenti** . Ciò significa che se la nostra applicazione utilizzasse la versione 4.9.175 di express in futuro, tutto il codice implementato in questa parte dovrebbe comunque funzionare senza apportare modifiche al codice. Al contrario, il futuro 5.0.0. la versione di express **potrebbe contenere** modifiche che potrebbero causare il mancato funzionamento della nostra applicazione.

## Web ed Express

Apportiamo le seguenti modifiche al file **index.js** :

```
const express = require('express')
const app = express();
let heroes = [
  ...
]
app.get('/', (request, response) => {
  response.send('<h1>Hello world</h1>')
```

```
})  
app.get('/api/heroes', (request, response) => {  
  response.json(heroes)  
})  
const PORT = 3002  
app.listen(PORT)  
console.log(`Server running on port ${PORT}`)
```

Avviate l'applicazione, prima chiudetela se lo era di già. Questa non è cambiata molto, inizialmente importiamo express, questa volta è una funzione utilizzata per creare un'applicazione express memorizzata nella variabile **app** :

```
const express = require('express')  
const app = express();
```

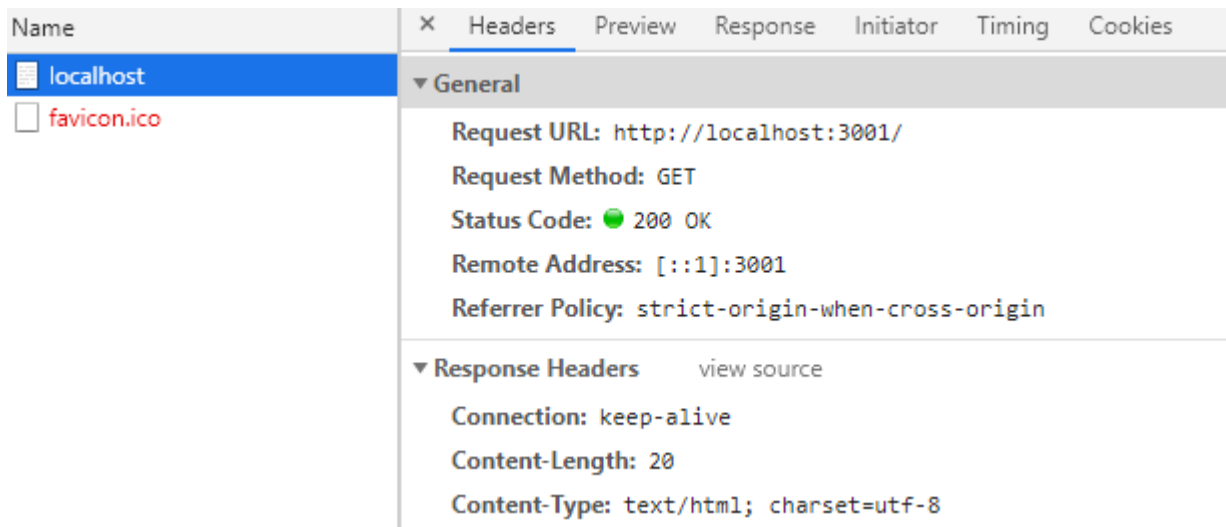
Successivamente, definiamo due percorsi per l'applicazione. Il primo definisce un gestore di eventi, che viene utilizzato per gestire le richieste HTTP GET fatte alla / root dell'applicazione :

```
app.get('/', (request, response) => {  
  response.send('<h1>Hello world</h1>')  
})
```

La funzione del gestore eventi accetta due parametri. Il primo parametro di **request** contiene tutte le informazioni della richiesta HTTP e il secondo parametro di **response** viene utilizzato per definire la modalità di risposta alla richiesta.

Nel nostro codice, la richiesta viene risposta utilizzando il metodo di **invio** dell'oggetto response. La chiamata al metodo fa sì che il server risponda alla richiesta HTTP inviando una risposta contenente la stringa **<h1>Hello World!</h1>**, che è stata passata al metodo **send** . Poiché il parametro è una stringa, express imposta automaticamente il valore dell'intestazione Content-Type su text / html . Il codice di stato della risposta è di default 200.

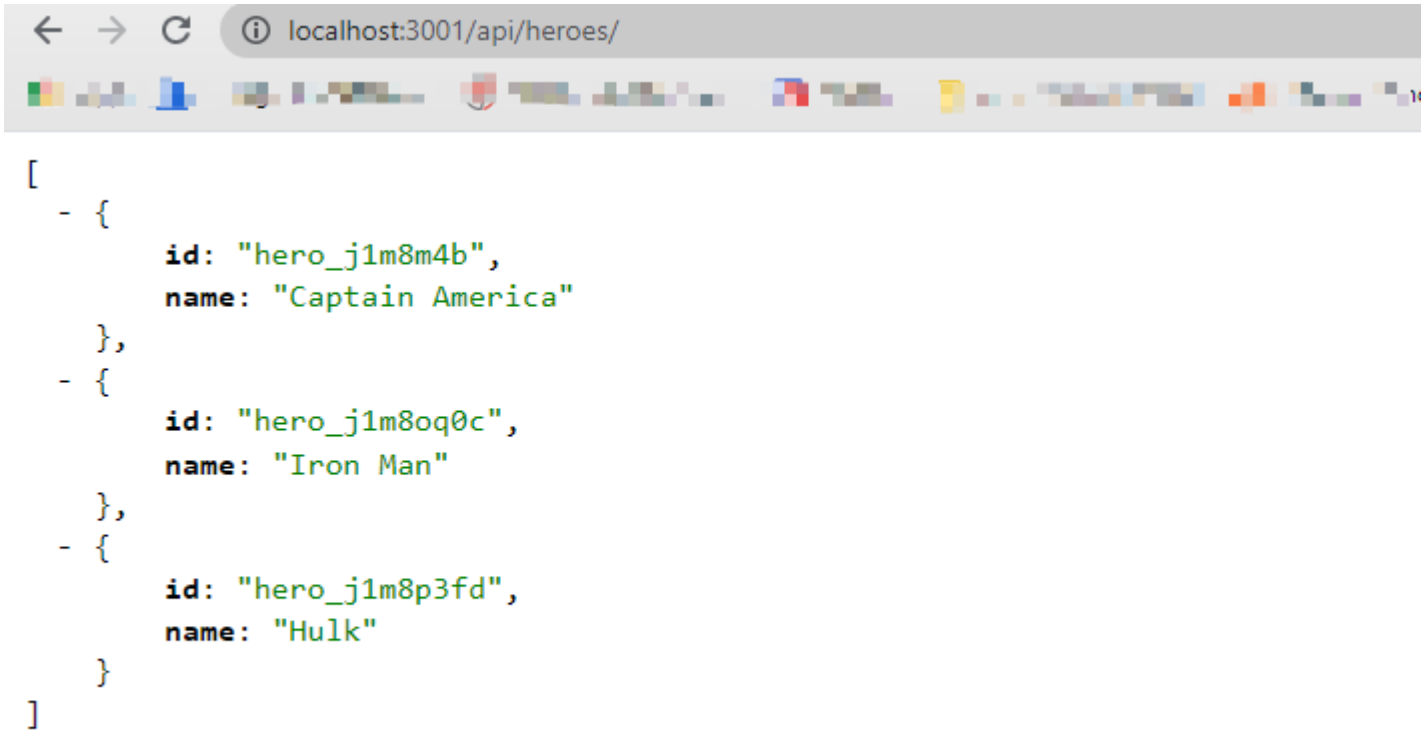
Possiamo verificarlo dalla scheda Rete o Network negli strumenti per sviluppatori:



La seconda route definisce un gestore di eventi, che gestisce le richieste HTTP GET effettuate al percorso degli heroes dell'applicazione:

```
app.get('/api/heroes', (request, response) => {  
  response.json(heroes)  
})
```

Alla **request** viene risposto con il metodo **json** dell'oggetto response. La chiamata al metodo invierà l' array di heroes che gli è stato passato come stringa formattata JSON. Express imposta automaticamente l' intestazione Content-Type con il valore appropriato di application / json .



```
[
  - {
    id: "hero_j1m8m4b",
    name: "Captain America"
  },
  - {
    id: "hero_j1m8oq0c",
    name: "Iron Man"
  },
  - {
    id: "hero_j1m8p3fd",
    name: "Hulk"
  }
]
```

Diamo una rapida occhiata ai dati inviati nel formato JSON.

Nella versione precedente in cui **usavamo** solo Node, dovevamo trasformare i dati nel formato JSON con il metodo **JSON.stringify** :

```
response.end(JSON.stringify(heroes))
```

Con express, questo non è più necessario, perché questa trasformazione avviene automaticamente.

Vale la pena notare che **JSON** è una stringa e non un oggetto JavaScript come il valore assegnato agli **heroes** .

## Nodemon

Quando apportiamo modifiche all'applicazione, per poterne vedere gli effetti nel browser dobbiamo prima chiuderla con **ctrl+c** e poi riavviarla. Un pò scomodo non trovate?

La soluzione a questo problema è **nodemon** :

nodemon controlla i file nella directory in cui è stato avviato lo stesso, e se qualche file cambia, egli riavvierà automaticamente l'applicazione del nodo.

Installiamo nodemon definendolo come una dipendenza di sviluppo con il comando:

```
npm install nodemon -D
```

controllate il file **package.json** :

```
...  
  "devDependencies": {  
    "nodemon": "^2.0.7"  
  }  
...
```

lo vedrete aggiunta nelle **devDependencies** .

Per dipendenze di sviluppo, ci riferiamo a strumenti che sono necessari solo durante lo sviluppo dell'applicazione, ad esempio per testare o riavviare automaticamente l'applicazione, come nodemon .

Queste dipendenze di sviluppo non sono necessarie quando l'applicazione viene eseguita in modalità di produzione sul server di produzione (ad esempio Heroku).

Possiamo avviare la nostra applicazione con nodemon in questo modo:

```
node_modules/.bin/nodemon index.js
```

Una volta che effettueremo ora le modifiche all'applicazione queste verranno salvate automaticamente, comunque , a differenza di quando lavoriamo con React, il browser deve comunque esser riavviato.

Il comando è abbastanza lungo per cui andiamo a definire un script npm nel file **package.json** :

```
...  
  "scripts": {  
    "start": "node index.js",  
    "dev": "nodemon index.js",  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  ...
```

Come potete notare non è necessario specificare il percorso completo di nodemon ma basta inserire **nodemon index.js**, **npm** sa dove crearlo a prescindere 😊

ora potete avviare l'applicazione con nodemon digitando:

```
npm run dev
```

In questo caso , a differenza degli script **start** e **test** dovete aggiungere anche **run** al comando.

## React con Node.js ed Express: Riepologo

Prima che il tutorial diventi prolisso mi fermo. Siamo riusciti a creare il backend grazie a NodeJs ed Express, nel prossimo [tutorial](#) continueremo ad implementarlo in quanto ancora incompleto. Vedremo come funzionano le REST API, i Middleware, ecc...