JavaScript Node.js

Async Programming in NodeJs

Url: https://github.com/obiSerra/laboratorio-nodejs

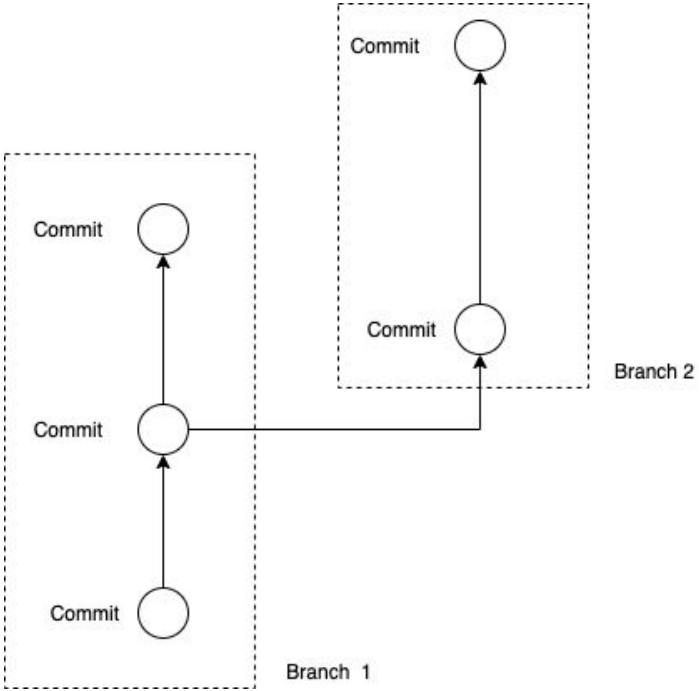
Breve riepilogo

- NodeJs e’ un runtime che permette di eseguire JavaScript “lato server”
- Utilizza V8 come alcuni browser per interpretare JS
- Npm e’ il packet manager default di Node; fornisce pacchetti e una CLI
- Esistono 2 tipi di module system in Node: CommonJs e ESM

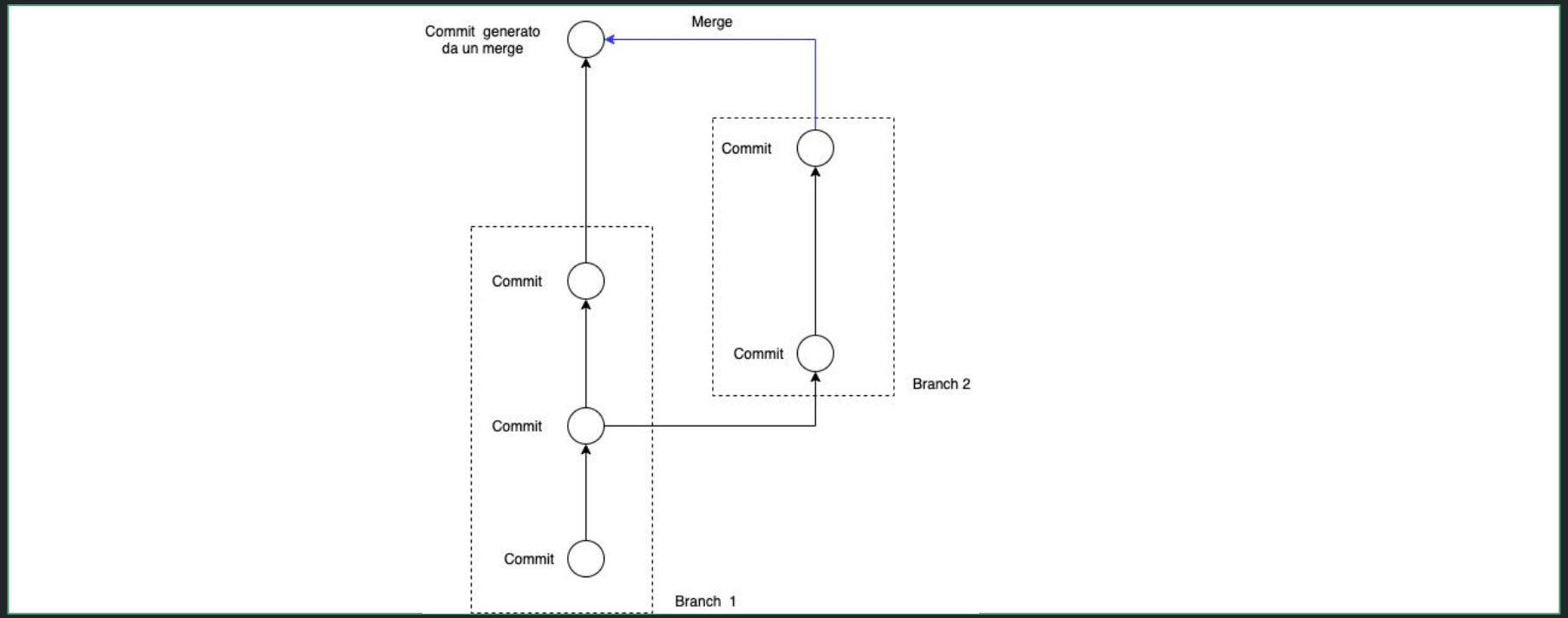
Git

- Sistema di versioning
- Sync directory locale con origin remoto
- Diversi rami (branches) composti da commit
- Push => carica i cambiamenti presenti in locale
- Pull => scarica i cambiamenti da origin
- Fetch => scarica le informazioni sui cambiamenti presenti su origin
- Merge => unisce due rami (o due versioni di un ramo). Solitamente e' automatico

Git 1



Git 2 - Merge


Commit generato
da un merge
Merge
Commit
Commit
Commit
Commit
Commit
Branch 2
Branch 1

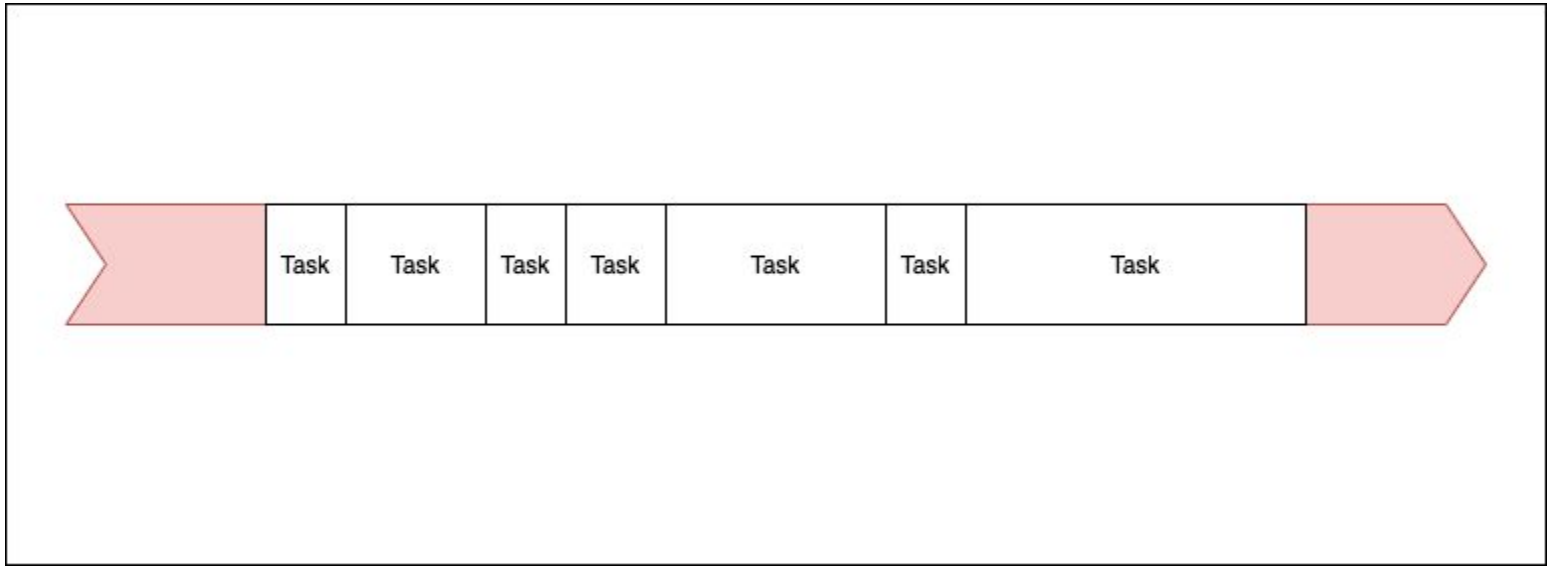
Correzione esercizio

Url: https://github.com/obiSerra/laboratorio-nodejs

Programmazione sincrona

- Le istruzioni vengono eseguite una dopo l’altra
- Es: lezione-2/esempi/sync.js
- Task lunghi o che utilizzano risorse esterne bloccano il processo

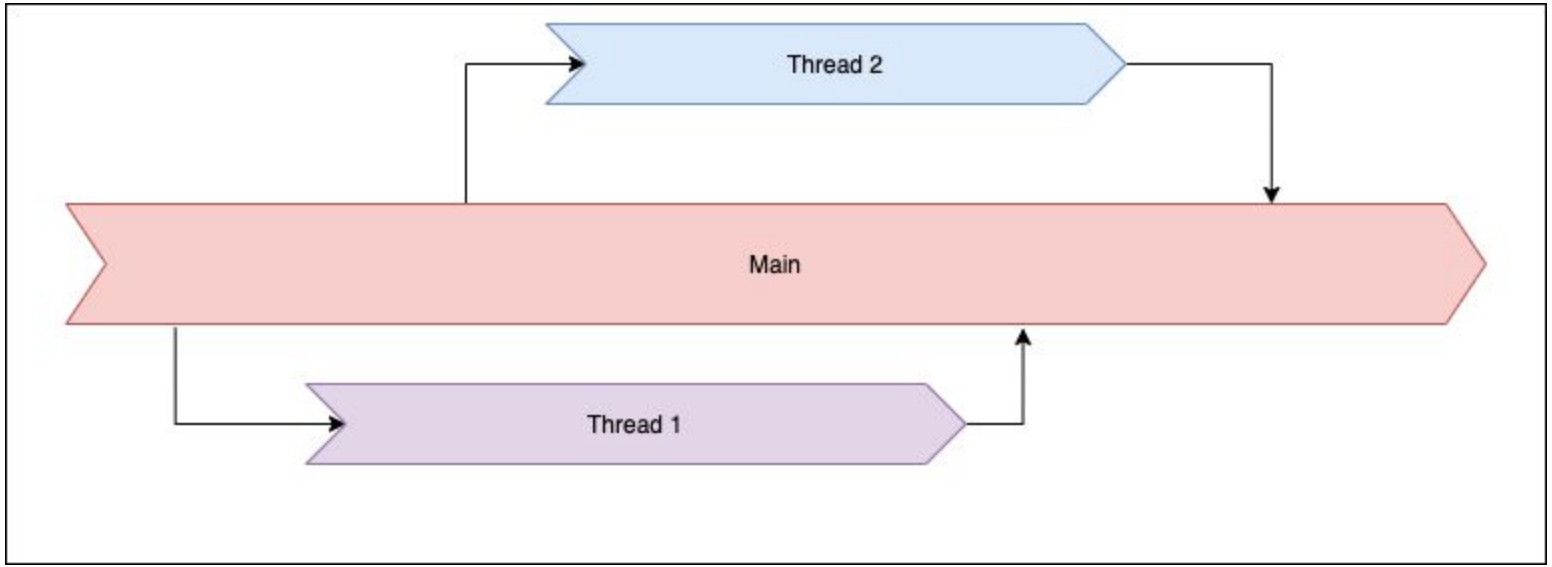
Programmazione sincrona



Threads

- In Computer Science un thread e’ una “sequenza” di istruzioni che possono essere eseguite all’interno di un processo
- Alcune runtime permettono di creare multipli thread (es lezione-2/multithread-py)
- Node invece e’ una runtime single-thread

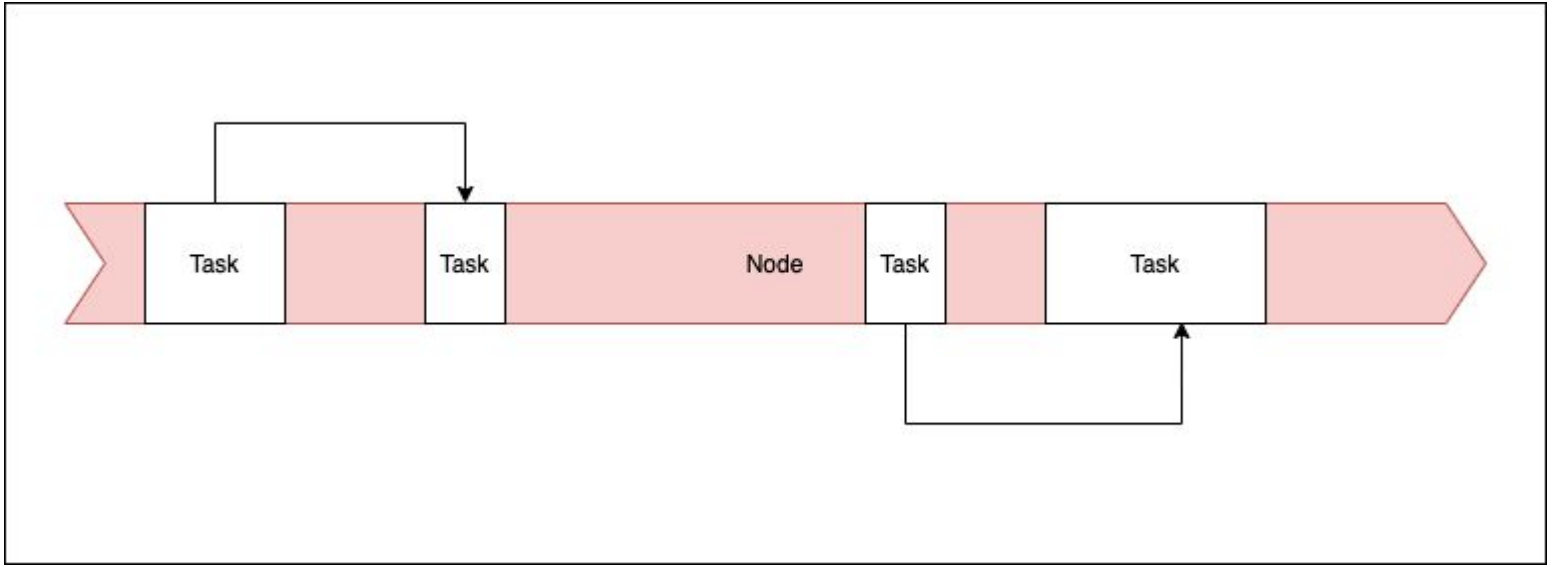
Multi-threads



NodeJs

- In Node non possiamo creare nuovi thread
- Libuv pero’ a basso livello utilizza una thread pool
- NodeJS e’ un sistema *event-driven* perche’ permette di agganciare azioni a particolare eventi (es: alla fine dell’esecuzione di un task)

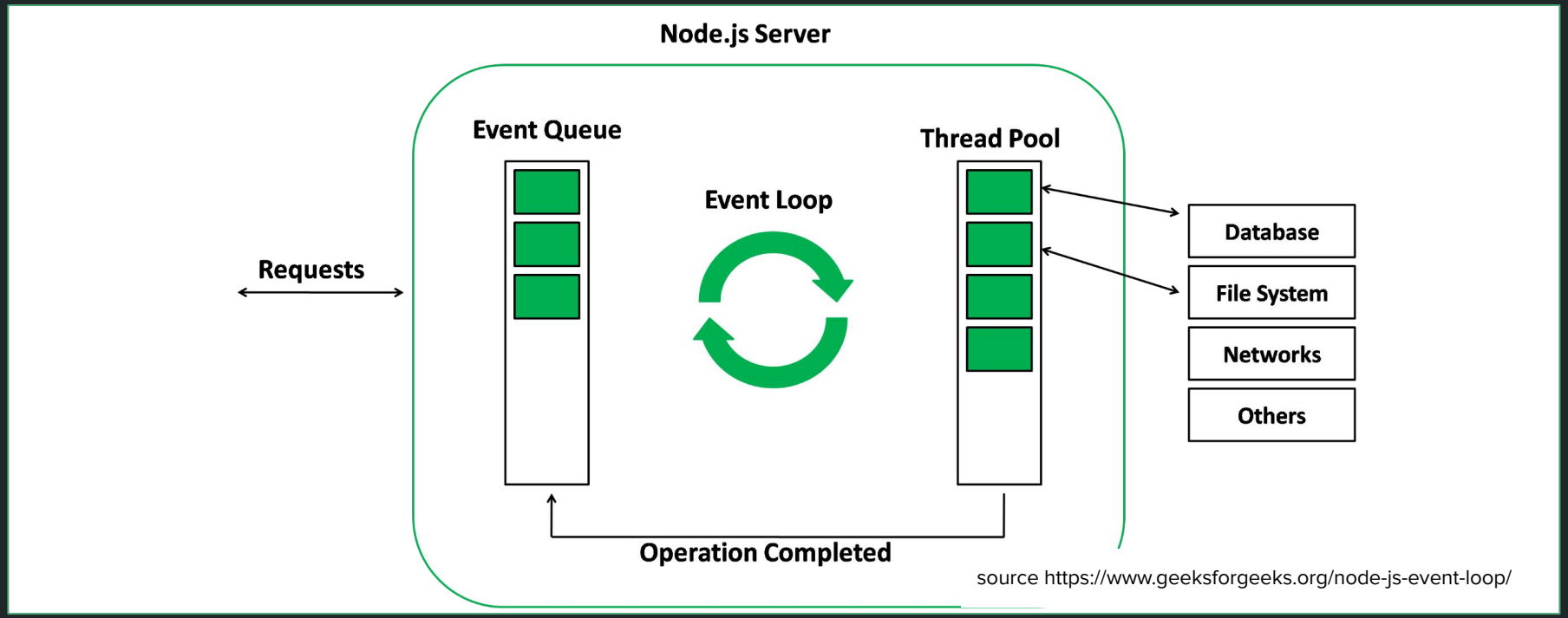
NodeJS



Event Loop

- Node (come JS lato browser) e’ basato su un meccanismo chiamato Event Loop
- L’ Event Loop e’ quello che permette a Node di eseguire *non-blocking operations*
- Si tratta di un loop eseguito continuamente e composto da diverse fasi

Node event loop



source https://www.geeksforgeeks.org/node-js-event-loop/

Callback

- *ref lezione-2/esempi/callbacks.js*
- Sono il metodo base per eseguire codice asincrono
- Possono essere sincrone o asincrone (attenzione a non mischiarle)
- In Node (ma non in JS) e’ convenzione passare la cb come ultimo parametro
- Attenzione alla *pyramid of doom*

Troppa teoria 0

Iniziamo a creare un tool per fare il crawling di un sito; chiamando

$ npm run crawl <url>

Voglio stampare sul termianale una lista di pagine a partire da quell'URL

Troppa teoria 1

- Creiamo un nuovo progetto “site_crawler”
- Struttura:
  - index.js
  - package.json
  - src
    - crawler.js
- ./src/crawler.js esporta una classe Crawler con un costruttore che prende 1 parametro (startingUrl)
- index.js controlla che venga passato un argomento quando viene chiamato; se e’ cosi’ instanzia Crawler passandogli l’argomento

Troppa teoria 1a

Possiamo usare il pacchetto URL presente all'interno di Node per validare il parametro

```
try {
    url = new URL(paramArgument)
} catch(e) {
   ...
}
```

Events

- *ref lezione-2/esempi/events.js*
- E’ possibile avere multipli eventi e multipli subscribers
- Possono essere usati estendendo la classe Events (https://nodejs.org/api/events.html)
- E’ possibile combinare callback ed eventi (*ref lezione-2/esempi/callback-events.js)*

Troppa teoria 2

- Usando il modulo *https* possiamo fare chiamate GET a un indirizzo (vedi https://github.com/obiSerra/laboratorio-nodejs/blob/main/lezione-2/esempi/callback-events.js)
- Aggiungiamo un metodo al crawler che scarichi la pagina richiesta
- Possiamo usare *response.on("end",...)* per sapere quando la pagina e' stata completamente scaricata

Troppa teoria 2a

Usando il pacchetto npm *node-html-parser* possiamo simulare un DOM sulla pagin scaricata per cercare tutti i link (a) presenti

```
htmlParser
    .parse(content)
    .querySelectorAll("a")
```

Promise

- *ref lezione-2/esempi/promise.js*
- Sono nate per risolvere il problema del *callback hell* e delle *pyramid of doom (ref lezione-2/esempi/pyramid.js)*
- Permettono il chaining di then
- Gestione degli errori con *.catch*
- Possono diventare complicate da gestire quando ci sono diverse “catene” in parallelo
- Promise.all fornisce un semplice modo gestire multiple risorse asincrone

Troppa teoria 3

- Modifichiamo il codice di *getPage* per ritornare una Promise
- Aggiungiamo una lista delle pagine visitate
- Validiamo gli *href* che abbiamo trovato verificando che abbiano lo stesso host del sito principale

Async/Await

- *ref lezione-2/esempi/async-await.js*
- Ultima aggiunta al panorama di JS (e Node)
- Permettono di scrivere codice asincrono come fosse sincrono
- E’ basato sulle promise, quindi supporta anche funzionalita’ come *Promise.all()*
- Gestione degli errori con *try/catch*
- *await* puo’ essere chiamato solo in una funzione dichiarata con *async*
- *await* blocca l’esecuzione del codice che segue rendendolo di fatto sincrono

Troppa teoria 4

Per visitare i link che abbiamo trovato possiamo implementare una coda:

- usiamo *setImmediate* per creare un loop
- se la coda e' piena, scarichiamo la pagina e aggiungiamo i nuovi link alla coda
- se la coda e' vuota e ci sono processi attivi non facciamo nulla
- se la coda e' vuota e non ci sono processi attivi, abbiamo finito

Note:

- Aggiungiamo un contatore per limitare il numero di chiamate massime

Troppa teoria 4a

Come ulteriori miglioramenti possiamo

- Registrare lo status code per ogni chiamata che facciamo
- Salvare il risultato finale (pagine visitate con relativo status) su un file
- Limitare il numero di operazioni asincrone parallele