NodeJS

Rotta verso le massime prestazioni

AHOY

gabriele fontana

github.com/gafreax

DI COSA PARLEREMO

Cos'è Node: Event Loop, Non-Blocking IO,

CPU-Bound / IO-Bound e colli di bottiglia in Node

Creazione di backend efficaci con fastify

Migliorare le perfomance per processi CPU-Bound

WORKER THREADS & CACHE

Node il nostro tesoro

NodeJS

Node.js è un ambiente di runtime JavaScript open-source, creato nel 2009, che ha guadagnato una vasta diffusione per la sua efficienza nella gestione di operazioni I/O e la sua architettura event-driven.

nodeJs

- Node.js è un runtime open-source basato su V8.
- Il suo design permette una esecuzione Non blocking I/O
- L'esecuzione base è single-threaded
- Molto Efficace per task memory based o IO
- Meno efficace task CPU intensive


I COLLI DI BOTTIGLIA
DI NODE

Colli di bottiglia di Node

Il nemico n°1:

- Blocco dell'Event Loop (CPU-Intensive task)

Gli altri nemici:

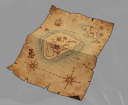
- Overhead dei framework
- Serializzazione/deserializzazione
- Esecuzioni async non sfruttate

PRIMA PARTE


FASTIFY

FASTIFY: INTRO

È un framework web Node.js progettato per prestazioni eccezionali e un'esperienza di sviluppo ottimale. Ideale per API REST e applicazioni web, offre velocità, basso overhead e un ecosistema di plugin flessibile.



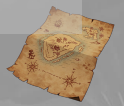
Fastify: inziamo!

```
# classic way
npm init
npm i fastify

# fastify cli
npx fastify-cli generate my-project
```

Fastify: Routes

L' implementazione di rotte è davvero molto semplice!

```
import Fastify from 'fastify'
const app = Fastify()

app.get('/hello', async (req,reply) => reply.send({msg: "ahoy"})
```

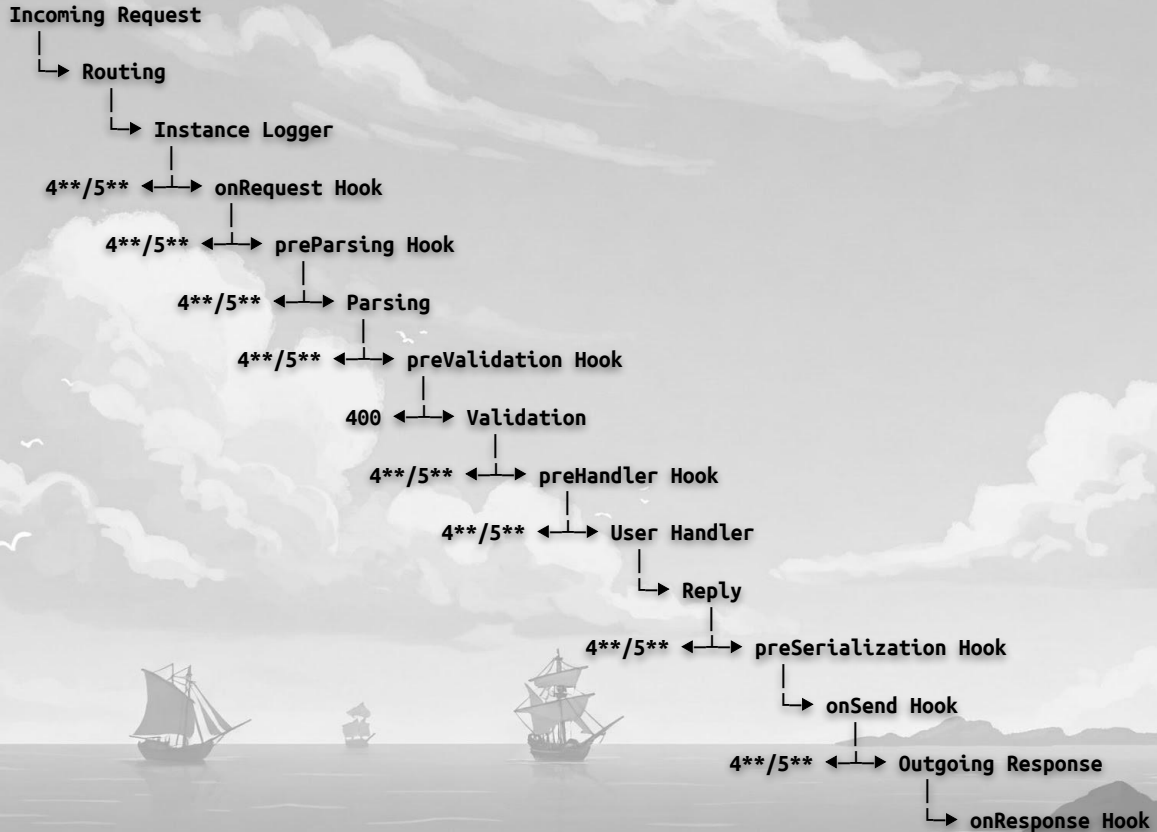


FASTIFY: PLUGIN

Fastify si basa sul concetto di plugin e decorator per estendere le funzionalità del nostro software

```
import Fastify from ‘fastify’
import fp from 'fastify-plugin'
const app = Fastify()
app.register(fp(async function pirate(instance) {
    instance.decorate(‘ahoy’, () => ({msg: ‘AHOY’})
}))
app.get('/pirate’, async (req,reply) => reply.send(app.ahoy())
```

FASTIFY: LIFECYCLE



FASTIFY: HOOKS!

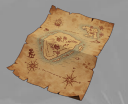
Possiamo agganciarci semplicemente ad una di queste fasi creando un hooks

```
app.addHook('onRequest', async (req, reply) => {
  const { headers } = req
  app.log.debug('Non ho ancora il body ma ho gli headers')
  if( headers['x-pirate'] ) {
    app.log.info('benvenuto')
  }
})
```

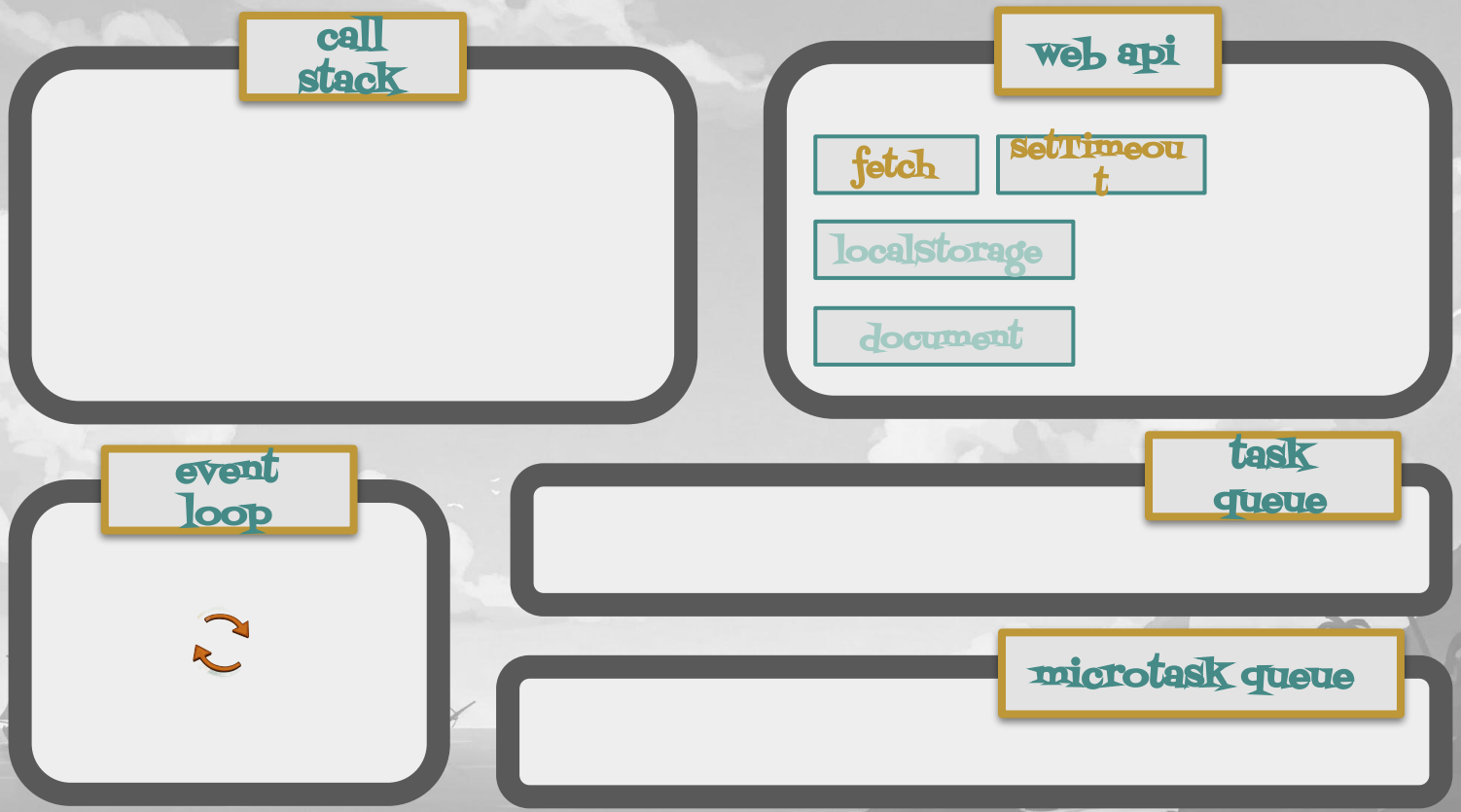

Event Loop

Event Loop

L'Event Loop è il meccanismo centrale di Node.js che gestisce e smista i compiti, eseguendo il codice JavaScript su un singolo thread e delegando le operazioni di I/O ad altri thread per non bloccarsi mai: Non Blocking IO



Event Loop: come funziona



event loop!

```
console.log(“a!”)
console.log(“b!”)
function logC(){
  console.log(“c!”)
}
function logCD(){
 logC()
 console.log(“d!”)
}
logCD()
```

console

call stack

web api

fetch

setTimeout

localstorage

document

event loop

task queue

microtask queue

event loop!

```
console.log(“a!”)
console.log(“b!”)
function logC(){
  console.log(“c!”)
}
function logCD(){
 logC()
 console.log(“d!”)
}
logCD()
```

call stack

```
console.log(“a!”)
```

web api

fetch

setTimeout

localstorage

document

task queue

microtask queue

event loop

console

```
a!
```

event loop!

```
console.log("a!")
console.log("b!")
function logC(){
  console.log("c!")
}
function logCD(){
 logC()
 console.log("d!")
}
logCD()
```

call stack

```
console.log("b!")
```

web api

fetch

setTimeout

localstorage

document

console

```
a!
b!
```

event loop

task queue

microtask queue

event loop!

```
console.log("a!")
console.log("b!")
function logC(){
  console.log("c!")
}
function logCD(){
 logC()
 console.log("d!")
}
logCD()
```

call stack

logCD()

web api

fetch

setTimeout

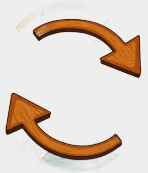
localstorage

document

console

a!
b!

event loop



task queue

microtask queue

event loop!

```
console.log("a!")
console.log("b!")
function logC(){
  console.log("c!")
}
function logCD(){
 logC()
 console.log("d!")
}
logCD()
```

call stack

logC()
logCD()

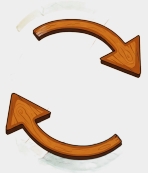
web api

fetch
setTimeout
localstorage
document

task queue

microtask queue

event loop



console

a!
b!

event loop!

```
console.log(“a!”)
console.log(“b!”)
function logC(){
  console.log(“c!”)
}
function logCD(){
 logC()
 console.log(“d!”)
}
logCD()
```

call stack

```
console.log(“c!”)
logC()
logCD()
```

web api

fetch

setTimeout

localstorage

document

console

a!
b!
c!

event loop

task queue

microtask queue

event loop!

```
console.log("a!")
console.log("b!")
function logC(){
  console.log("c!")
}
function logCD(){
 logC()
 console.log("d!")
}
logCD()
```

call stack

```
console.log("d!")

logCD()
```

web api

fetch

setTimeout

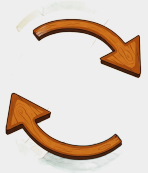
localstorage

document

console

```
a!
b!
c!
d!
```

event loop



task queue

microtask queue


async/await

event loop: async!

```
function fn(n) {
 fetch(url,  {
   method:post,
   body:`{stat:${n}}`
 }).then(() =>
  console.log("done")
 )
 console.log("ret")
}
fn(3)
fn(4)
```

console

ret
done

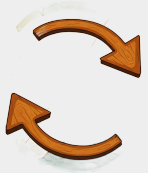
call stack

```
console.log("ret")
fn(3)
```

web api

fetch
setTimeout
localstorage
document

event loop



task queue

fetch

microtask queue

```
console.log("done")
```


IMPATTO DI PROCESI
CPU-INTENSIVE

event loop: block it!

```
function longTask() {
 let c = 0
 for(letj=0;j<1e9;j++)
   c++
 console.log("done!")
}
function important() {
  console.log("imp!")
}
longTask()
important()
```

console

call stack

web api

fetch

setTimeout

localstorage

document

event loop

task queue

microtask queue

event loop: block it!

```
function longTask() {
 let c = 0
 for(letj=0;j<1e9;j++)
   c++
 console.log("done!")
}
function important() {
  console.log("imp!")
}
longTask()
important()
```

call stack

longTask()

web api

fetch

setTimeout

localstorage

document

console

event loop

task queue

microtask queue

event loop: block it!

```
function longTask() {
 let c = 0
 for(letj=0;j<1e9;j++)
   c++
 console.log("done!")
}
function important() {
  console.log("imp!")
}
longTask()
important()
```

console

call stack

longTask()

web api

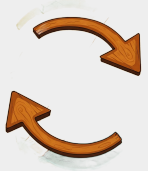
fetch

setTimeout

localstorage

document

event loop



task queue

microtask queue


3 HOURS LATER...
Zzzz..

event loop: block it!

```
function longTask() {
 let c = 0
 for(letj=0;j<1e9;j++)
   c++
 console.log("done!")
}
function important() {
  console.log("imp!")
}
longTask()
important()
```

call stack

```
console.log("done!")
longTask()
```

web api

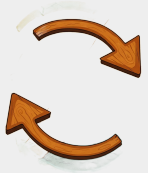
fetch

setTimeout

localstorage

document

event loop



task queue

microtask queue

console

done!

event loop: block it!

```
function longTask() {
 let c = 0
 for(letj=0;j<1e9;j++)
   c++
 console.log("done!")
}
function important() {
  console.log("imp!")
}
longTask()
important()
```

call stack

```
console.log("imp!")
important()
```

web api

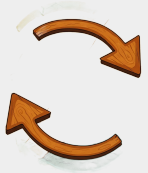
fetch

setTimeout

localstorage

document

event loop



task queue

microtask queue

console

```
done!
imp!
```

Soluzioni per task CPU intensive

All'arrembaggio con i Worker Threads

Worker Threads

I Worker Threads sono un modo per sfruttare al massimo le CPU multi-core, migliorando le performance.

Worker Threads

Permettono di eseguire operazioni intensive sulla CPU in background senza bloccare il thread principale, garantendo che l'applicazione rimanga reattiva e performante.

USIAMO LE SCORTE: LA CACHE

CACHE

La cache è uno strumento fondamentale per ottimizzare le prestazioni: è una memoria temporanea ad alta velocità che memorizza dati usati frequentemente per accelerare l'accesso futuro

CACHE

Per ottimizzare l'utilizzo delle risorse e ottenere sempre il massimo delle prestazioni, non sprecare risorse dovremo attivare anche una strategia di caching, evitando di processare ciò che è già stato elaborato


JSON SCHEMA: IL
NOSTRO SCUDO

JSON-SCHEMA VALIDATION

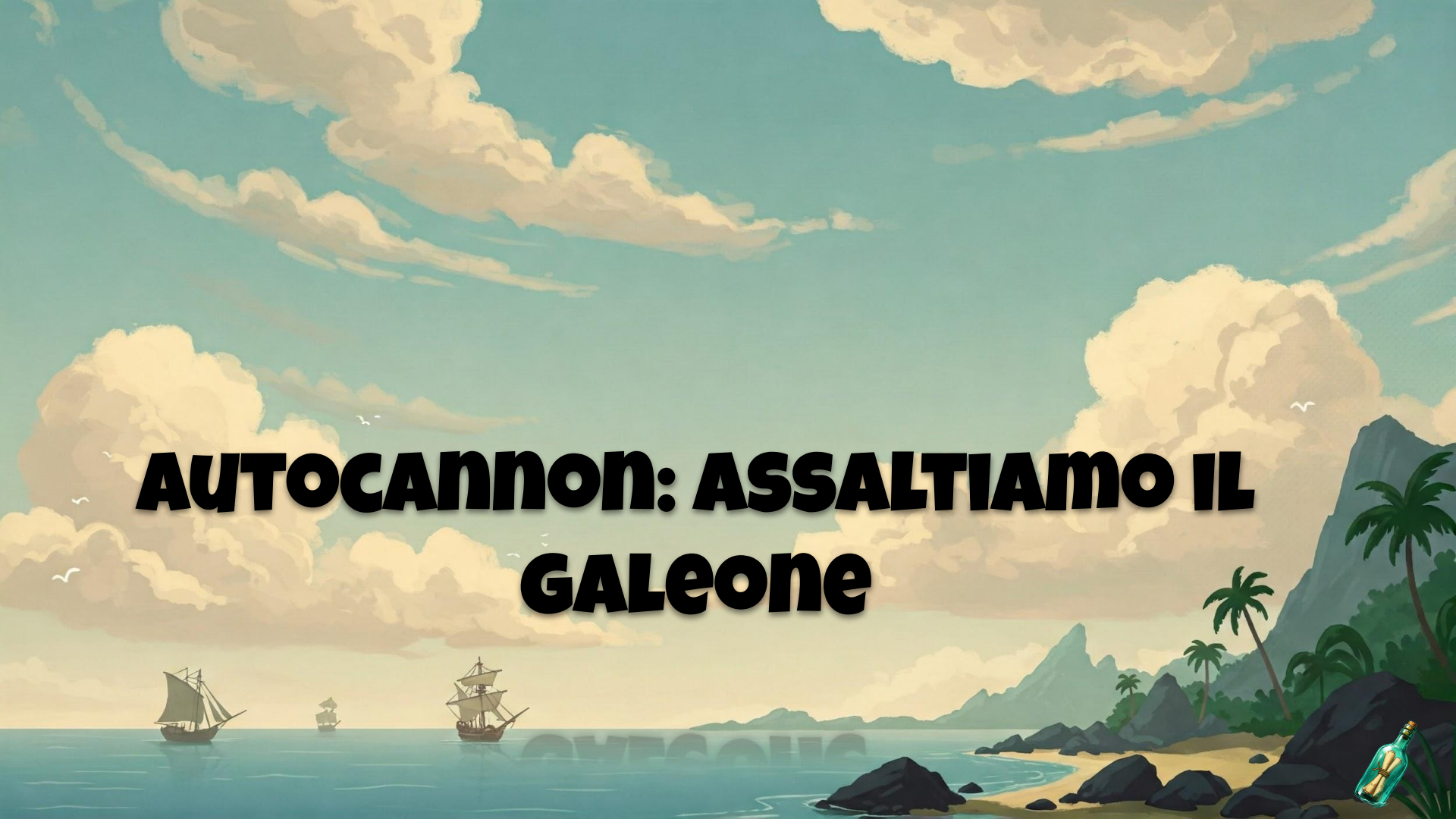
Un tema fondamentale nella creazione di servizi REST oggi, è quello della serializzazione e deserializzazione, e della validazione dei JSON gestiti

JSON-SCHEMA VALIDATION

Utilizzare un sistema di deserializzazione/serializzazione solido e veloce ci permette di ottenere performance e robustezza del nostro software

SECONDA PARTE

AUTOCANNON: ASSALTIAMO IL GALEONE



Autocannon Benchmark

È fondamentale quando si progetta un backend monitorare le sue performance la loro evoluzione.

```
$ npx autocannon http://localhost:3123/
```

PISCINA

PISCINA

È una libreria, molto performante, che permette di accedere ad una semplice astrazione per la gestione dei worker threads

PISCINA

```
// index.js
const path = require('path');
const Piscina = require('piscina');
const piscina = new Piscina({
  filename: path.resolve(__dirname, 'worker.js')
});

(async function() {
  const result = await piscina.run({ a: 4, b: 6 })
  console.log(result);  // Stampa 10
})()

// In worker.js:
module.exports = ({ a, b }) =>  a + b }
```


Cache: node-cache

node-cache

È una libreria, che permette di implementare una semplice strategia di cache interna.
Molto utile per prototipare, per la produzione è meglio rivolgersi a REDIS

node-cache

```
import NodeCache from "node-cache"
const myCache = new NodeCache()

myCache.set(‘Guybrush’, {
    name: 'Guybrush Threepwood',
    age: 27,
    image: 'guybrush.png',
})

const r = myCache.get(‘Guybrush’)
```

Schema validation con Fastify

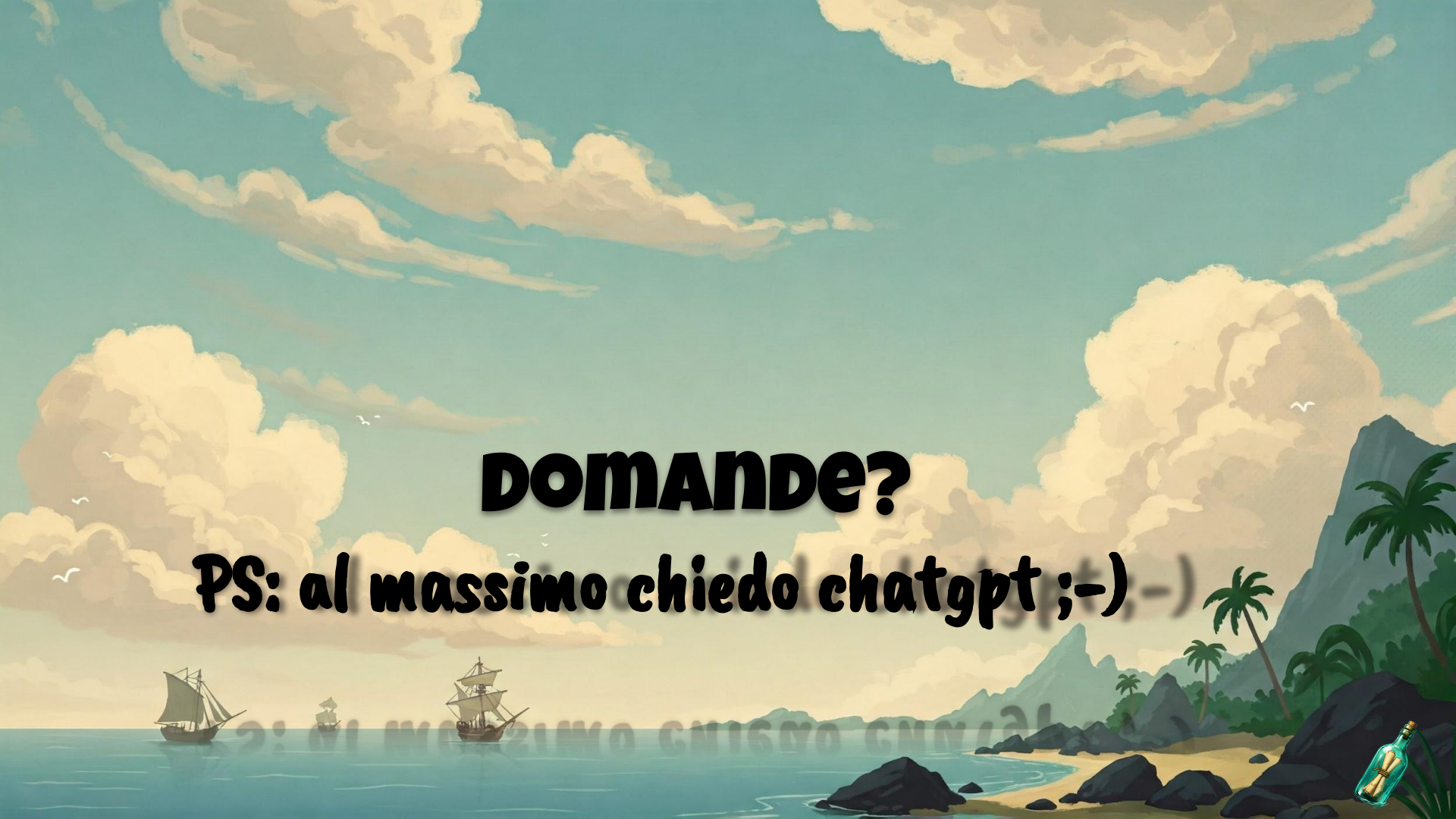
SCHEMA VALIDATION

Validare uno schema quando il content/type è un JSON diventa fondamentale per garantire robustezza al nostro backend . Inoltre agendo in fase di deserializzazione/serializzazione migliora anche le performance

SCHEMA VALIDATION

```
const pirateSchema = {
 $id: 'pirate',
 type: 'object',
 properties: {
   name: { type: 'string' }
}

app.post('/pirate', { schema: { body: pirateSchema } }, async (req, reply) => {
 reply.send({ msg: “Arrr!” })
})
```


TL;DR:
NON BLOCCARE L'EVENT LOOP
USARE LA CACHE
MONITORARE LE PERFORMANCE
CREARE BE ROBUSTI


DOMANDE?
PS: al massimo chiedo chatgpt ;-)

GRAZIE MILLE!