Node.js

9.0 - Introduzione a Node.js

Node.js nasce con l'obiettivo di permettere ai programmatori JavaScript di sviluppare anche applicazioni lato server. Tradizionalmente, nei sistemi web abbiamo una separazione netta tra i linguaggi: JavaScript sul client, mentre sul server si utilizzano linguaggi come Java, PHP o C#, con SQL o MQL per l'interazione con i database. Questa divisione richiede competenze diverse e limita i programmatori JavaScript al solo lato client.

Con l'evoluzione verso i Web Service, in cui il lato client potrebbe anche non essere sviluppato direttamente, Node.js offre la possibilità di utilizzare JavaScript anche sul lato server, permettendo di manipolare i dati JSON con lo stesso linguaggio in cui questi sono nati. Node.js può essere facilmente scaricato dal sito ufficiale (https://nodejs.org/it/), con l'ultima versione disponibile che è la 22.14.0 (2025). Durante la procedura di download, è consigliato installare anche le utilità supplementari, in particolare "npm", che consente di installare pacchetti aggiuntivi.

9.1 - Programmazione a Eventi in Node.js

I programmi Node.js sono fortemente basati sulla programmazione a eventi, implementata in JavaScript attraverso le funzioni di callback. Anche le elaborazioni più complesse dovrebbero essere organizzate con chiamate indirette a funzioni di callback, generando di fatto eventi interni. Node.js gestisce diverse code di eventi con diversa priorità attraverso il cosiddetto "Event Loop", che ha il compito di:

1. Prendere un evento dalla coda non vuota a più alta priorità
2. Chiamare la funzione di callback associata
3. Ripetere questo processo all'infinito.

Quando arriva una richiesta ad una porta TCP, questa viene trasformata in evento, accodato in una coda di eventi. Quando è il turno dell'evento, viene processato e la funzione di callback corrispondente viene chiamata. In questo modo, il sistema può gestire richieste multiple su porte diverse.

9.2 - Oggetti Globali in Node.js

Un programma per Node.js non lavora nel browser, quindi non è associato ad una finestra e l'oggetto Window non esiste. Il nuovo oggetto di contesto si chiama `global`. Un programma JavaScript per Node.js ha il compito di attivare un server, cioè mettersi in ascolto su una porta TCP e rispondere alle richieste entranti. Inoltre, può gestire in modo nativo il protocollo HTTP (e HTTPS).

L'oggetto global fornisce strumenti utili per creare il server e gestire il processo di comunicazione. Tra gli oggetti offerti da `global` troviamo:

- `process` : consente di gestire il processo di elaborazione in corso
- `console` : consente di scrivere messaggi (di debug) sulla console associata al processo
- `__dirname` : la cartella da cui il processo è stato lanciato
- `__filename` : il file con il codice JavaScript del processo

Global offre anche diverse classi (costruttori) come:

- `Buffer` : gestisce gli array, usato in particolare per ricevere gli argomenti sulla linea di comando
- `URL` : consente di gestire gli URL
- `URLSearchParams` : gestisce i parametri di ricerca dell'URL (la query string del metodo GET)
- `TextDecoder` : effettua il decoding dei caratteri di un testo
- `TextEncoder` : effettua l'encoding di un testo

Tra i metodi offerti da global troviamo:

- `require()` : indica i moduli da importare
- `setTimeout()` : imposta un timeout per chiamare una funzione di callback
- `setInterval()` : schedula la chiamata ripetuta di una funzione di callback ad intervalli regolari
- `setImmediate()` : inserisce la funzione di callback specificata nella coda degli eventi
- `clearTimeout()` : cancella un timeout precedentemente impostato
- `clearInterval()` : cancella un interval precedentemente schedulato
- `clearImmediate()` : rimuove dalla coda degli eventi una richiesta di chiamare una funzione di callback

9.3 - Esecuzione di un Programma Node.js

Per eseguire un programma Node.js, ad esempio "server.js", si utilizza il comando "node" seguito dal nome del file:

```
node server.js
```

Essendo un server, il processo rimane attivo e aspetta di servire richieste. Per interrompere l'esecuzione si usa la combinazione di tasti Ctrl + C.

Esempio di Codice per un Server HTTP

Un esempio di codice per creare un server HTTP in Node.js si compone di diverse parti. Nella parte iniziale, occorre specificare i moduli che verranno utilizzati:

```
const http = require('http')
const url = require('url')
const hostname = '127.0.0.1'
const port = '8080'
```

Qui vengono importati i moduli http (per creare server http) e url (per manipolare gli URL), e vengono definite due costanti: l'indirizzo IP dell'host e la porta TCP.

Successivamente, viene creato il server con http.createServer, passando come parametro la funzione di callback da chiamare quando arriva una richiesta:

```
const server = http.createServer(function (req, res) {
    // ...
});
server.listen(port, hostname, function () {
    console.log('Server running at http://${hostname}:${port}/')
})
```

Il server viene associato all'host e alla porta con il metodo server.listen(), che accetta anche una funzione di callback chiamata quando il server è effettivamente in ascolto.

Gestione delle Richieste HTTP

La funzione di callback del server riceve due parametri:

- req: oggetto che descrive la richiesta HTTP (classe http.ClientRequest)
- res: oggetto che gestisce la risposta (classe http.ServerResponse)

Per ottenere i parametri dall'URL della richiesta:

```
var myurl = new url.URL("http://" + req.headers.host + req.url);
var params = myurl.searchParams;
var param_a = "";
if (params.has("a"))
    param_a = params.get("a");
```

Vogliamo ottenere il paramentro `<<a>>` dall'URL della richiesta, `req.url` ci fornisce l'URL senza dominio e protocollo (ovvero da `<</>>` in poi), dato che URL richede l'URL completo possimo ottenere anche il domnio usando `req.headers.host` .
Il campo myurl.searchParams è un oggetto che gestisce i parametri della ricerca. Il metodo params.has verifica la presenza del parametro richiesto, mentre params.get restituisce il valore del parametro.

Per preparare l'output, si può creare un documento JSON:

```
var o = new Object();
o.param_a = param_a;
var l = new Object();
for (const [name, value] of params) {
    l[name] = value;
}
o.list = l;
```

Infine, si predispone l'header della risposta e si invia:

```
res.statusCode = 200
res.setHeader('Content-Type', 'application/json')
res.end(JSON.stringify(o));
```

Per invocare possoiamo quindi connetterci al server passando come parametri `a=b` e `c=d` :
`http://127.0.0.1:8080/?a=b&c=d`

Debugging in Node.js

Si può fare debugging del codice JavaScript attivando Node.js in modalità di debug. Viene aperta una porta TCP specifica scelta a caso (per esempio la 9229) che dei tool esterni possono contattare:

```
node --inspect server.js
```

Per utilizzare il debugger, si può aprire Google Chrome e nella barra degli indirizzi scrivere:

```
chrome://inspect
```

Viene avviato lo strumento di debugging di Chrome. Cliccando su "Open Dedicated Debugger for Node" si apre la finestra del debugger. Per aprire il file sorgente, si preme Ctrl + P e si sceglie il file di interesse. A questo punto le funzionalità del debugger sono attive e si possono impostare breakpoint.

Se la porta 9229 risulta occupata, si può cambiare con l'opzione --inspect-port:

```
node --inspect-port=9228 --inspect server.js
```

9.4 - Node.js e MongoDB

Node.js può essere integrato con MongoDB per creare applicazioni web complete che utilizzano JavaScript sia sul lato client che sul lato server. Questa integrazione permette di lavorare con dati in formato JSON in modo nativo, dato che sia Node.js che MongoDB sono ottimizzati per questo formato. L'approccio event-driven di Node.js si sposa bene con le operazioni asincrone necessarie per interagire con un database come MongoDB.

Installazione del Driver MongoDB per Node.js

Per far connettere Node.js a MongoDB, è necessario installare il driver appropriato. Il driver ufficiale può essere installato utilizzando npm (Node Package Manager) con il seguente comando:

```
npm install mongodb --save
```

Il flag `--save` aggiunge automaticamente il pacchetto alle dipendenze nel file package.json del progetto.

Configurazione della Connessione a MongoDB

Per avviare una connessione a MongoDB da un'applicazione Node.js, è necessario importare il modulo mongodb e configurare alcuni parametri di base:

```
const mongo = require('mongodb')
const db_name = "MyDB_test";
const collection_name = "mycollection";
const db_url = "mongodb://localhost:27017";
```

In questo esempio, si definiscono:

- L'URL di connessione al database MongoDB (`db_url`), che in questo caso punta a un'istanza locale sulla porta predefinita 27017
- Il nome del database a cui connettersi (`db_name`)
- Il nome della collezione che verrà utilizzata (`collection_name`)

Creazione del Client MongoDB

Dopo aver importato il modulo e definito i parametri di base, è necessario creare un client MongoDB:

```
var client_config = {
    useUnifiedTopology: true,
    useNewUrlParser: true
};
const client = new mongo.MongoClient(db_url, client_config)
```

Il client viene configurato con due opzioni importanti:

- `useUnifiedTopology` : abilita il nuovo motore di topologia unificata
- `useNewUrlParser` : utilizza il nuovo parser dell'URL di connessione

Queste opzioni sono consigliate per evitare warning di deprecazione e utilizzare le funzionalità più recenti.

Connessione al Database MongoDB

Una volta creato il client, è possibile connettersi al database utilizzando il metodo `connect()` . Poiché le operazioni di database sono asincrone, è necessario utilizzare il pattern delle callback o, come nell'esempio seguente, le funzioni asincrone con async/await:

```
client.connect(async function(err) {
    if(err) {
        console.log("Error connecting to MongoDB");
        throw err;
    }
    // Qui si accede al database e si eseguono le operazioni
})
```

La funzione di callback viene eseguita dopo il tentativo di connessione. La parola chiave `async` indica che la funzione è asincrona, il che permette di utilizzare `await` all'interno del suo corpo per attendere il completamento di altre operazioni asincrone.

Accesso al Database e alle Collezioni

Una volta stabilita la connessione, è possibile accedere al database e alle collezioni desiderate:

```
db = await client.db(db_name);
var collection = await db.collection(collection_name);
```

Il metodo `db()` restituisce un riferimento al database specificato, mentre `collection()` restituisce un riferimento alla collezione desiderata. Il prefisso `await` assicura che l'operazione venga completata prima di procedere con il codice successivo.

Esecuzione di Operazioni CRUD su MongoDB

Con il riferimento alla collezione, è possibile eseguire varie operazioni CRUD (Create, Read, Update, Delete):

Inserimento di un documento

```
// Creazione di un oggetto da inserire
var o = new Object();
for (const [name, value] of params) {
    o[name] = value;
}

// Inserimento nel database
var r = await collection.insertOne(o);
if(r.result.ok == 1) {
    output_text = JSON.stringify({inserted: 1});
    MIME = MIME_json;
}
```

Questo codice crea un nuovo oggetto basato sui parametri ricevuti da una richiesta HTTP e lo inserisce nella collezione. Il risultato dell'operazione viene verificato per confermare che l'inserimento sia avvenuto con successo.

Lettura di documenti

```
// Recupero dei primi 10 documenti
const docs = await collection.find().limit(10).toArray();
// Conteggio totale dei documenti nella collezione
const total = await collection.find().count();

// Creazione dell'oggetto di risposta
var o = new Object();
o.Total = total;
o.Selected = 10;
o.docs = docs;
output_text = JSON.stringify(o);
```

Questo codice esegue due query: una per recuperare i primi 10 documenti della collezione, trasformandoli in un array, e una per contare il numero totale di documenti. I risultati vengono quindi combinati in un oggetto di risposta.

Chiusura della Connessione

Al termine delle operazioni, è importante chiudere la connessione al database per liberare risorse e prevenire problemi:

```
client.close();
```

Questo metodo chiude la connessione al database MongoDB, evitando di mantenere troppe connessioni aperte.

Esempio Completo di Server Node.js con MongoDB

Vediamo un esempio completo di un server Node.js che si integra con MongoDB per gestire due tipi di richieste:

1. `/insert?f1=v1&f2=v2&...` : inserisce un nuovo documento nella collezione con i campi e i valori specificati come parametri della query
2. `/list` : restituisce un documento JSON contenente i primi 10 documenti nella collezione e il conteggio totale

```
const http = require('http')
const url = require('url')
const mongo = require('mongodb')
const hostname = '127.0.0.1'
const port = '8080'
const MIME_json = "application/json";
const MIME_text = "text/plain";
var MIME = MIME_text;
var output_text = "";
const db_name = "MyDB_test";
const collection_name = "mycollection";
const db_url = "mongodb://localhost:27017";

const server = http.createServer(function (req, res) {
    console.log("Request Received\n");
    var myurl = new url.URL("http://" + req.headers.host + req.url);
    var params = myurl.searchParams;

    var client_config = {
        useUnifiedTopology: true,
        useNewUrlParser: true
    };
    const client = new mongo.MongoClient(db_url, client_config)

    // Connect to the db
    client.connect(async function(err) {
        if(err) {
            console.log("Error connecting to MongoDB");
            throw err;
        }
        db = await client.db(db_name);

        // Handle /insert request
        if(myurl.pathname == "/insert") {
            var o = new Object();
            for (const [name, value] of params) {
                o[name] = value;
            }

            var collection = await db.collection(collection_name);
            var r = await collection.insertOne(o);
            if(r.result.ok == 1) {
                output_text = JSON.stringify({inserted: 1});
```

```
                MIME = MIME_json;
            }
        }

        // Handle /list request
        if(myurl.pathname == "/list") {
            var collection = await db.collection(collection_name);
            const docs = await collection.find().limit(10).toArray();
            const total = await collection.find().count();

            var o = new Object();
            o.Total = total;
            o.Selected = 10;
            o.docs = docs;
            output_text = JSON.stringify(o);
            MIME = MIME_json;
        }

        // Close connection and send response
        client.close();
        res.statusCode = 200
        res.setHeader('Content-Type', MIME)
        res.end(output_text);
    });
});

server.listen(port, hostname, function () {
    console.log(`Server running at http://${hostname}:${port}/`)
})
```