Node.js - Introduction

What is Node.js?

Node.js was developed by Ryan Dahl in 2009. The definition of Node.js as is as follows:

Node.js is a platform built on Chrome's JavaScript runtime

for easily building fast and scalable network applications.

Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient,

perfect for data-intensive real-time applications that run across distributed devices.

What is Node.js?

Node.js is an open source, cross-platform runtime environment for developing server-side and networking applications.

Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.

Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Features of Node.js

1. **Asynchronous and Event Driven** − All APIs of Node.js library are asynchronous, non-blocking.
2. **Very Fast** − Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
3. **Single Threaded but Highly Scalable** − Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable and the same program can provide service to a much larger number of requests than traditional servers.
4. **No Buffering** − Node.js applications never buffer any data. These applications simply output the data in chunks.
5. **License** − Node.js is released under the MIT license.

Who Uses Node.js?

- eBay
- GoDaddy
- PayPal
- Uber
- Yahoo
- And a lot more…

Where to Use Node.js?

Following are the areas where Node.js is proving itself:

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications

Where Not to Use Node.js?

It is not advisable to use Node.js for CPU intensive applications.

Node.js - REPL Terminal

Node.js - REPL Terminal

REPL stands for:

Read **E**val **P**rint **L**oop

It represents a computer environment like a Windows console or Unix/Linux shell where a command is entered and the system responds with an output in an interactive mode.

Node.js - REPL Terminal

Node.js or **Node** comes bundled with a REPL environment. It performs the following tasks:

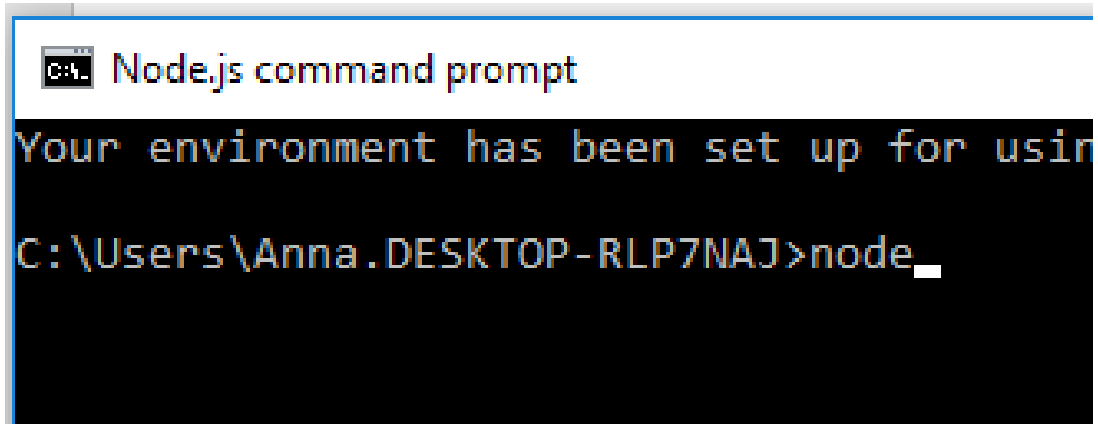
- **Read** – Reads user's input, parses the input into JavaScript data-structure, and stores in memory.
- **Eval** – Takes and evaluates the data structure.
- **Print** – Prints the result.
- **Loop** – Loops the above command until the user presses **ctrl-c** twice.

The REPL feature of Node is very useful in experimenting with Node.js codes and to debug JavaScript codes.

Starting REPL

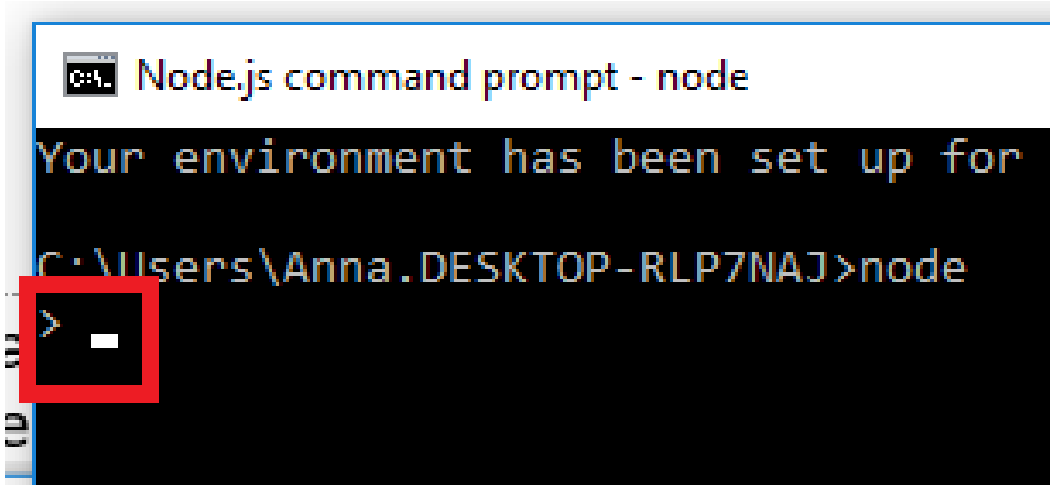
Step 1

REPL can be started by simply running node on shell/console without any arguments as follows:



Step 2

You will see the REPL Command prompt > where you can type any Node.js



Simple Expression

Let's try a simple mathematics at the Node.js REPL command prompt:

Node.js command prompt - node

```
Your environment has been set up for

C:\Users\Anna.DESKTOP-RLP7NAJ>node
> 3+4
7
> 2%1
0
> 7*6
42
>
```

Use Variables

You can make use variables to store values like any conventional script.

- If **var** keyword is not used, then the value is stored in the variable and printed.
- if **var** keyword is used, then the value is stored but not printed
- You can print variables using **console.log()**.

```
> var a=1
undefined
> b=2
2
> console.log(a)
1
undefined
> console.log(b)
2
undefined
> console.log(a+b)
3
undefined
>
```

Underscore Variable

You can use underscore **(_)** to get the last result:

```
> 2+8
10
> _
10
```

Multiline Expression

Node REPL supports multiline expression similar to JavaScript.
... comes automatically when you press Enter after the opening bracket.

Node.js command prompt - node

```
Your environment has been set up for

C:\Users\Anna.DESKTOP-RLP7NAJ>node
> var a=1
undefined
> while(a<5){
... console.log(++a);
... }
2
3
4
5
undefined
>
```

REPL Commands

.break – exit from multiline expression.

.clear – exit from multiline expression.

.save *filename* – save the current Node REPL session to a file.

.load *filename* – load file content in current Node REPL session.

REPL Commands

ctrl + c – terminate the current command.

ctrl + c twice – terminate the Node REPL.

ctrl + d – terminate the Node REPL.

Up/Down Keys – see command history

REPL Commands

tab Keys – list of current commands.

```
>
Array                                 Boolean                               Date                                  Error
EvalError                             Function                              Infinity                              JSON
Math                                  NaN                                   Number                                Object
RangeError                            ReferenceError                        RegExp                                String
SyntaxError                           TypeError                             URIError                              decodeURI
decodeURIComponent                    encodeURI                             encodeURIComponent                    eval
isFinite                              isNaN                                 parseFloat                            parseInt
undefined

ArrayBuffer                           Atomics                               Buffer                                COUNTER_HTTP_CLIENT_REQUEST
COUNTER_HTTP_CLIENT_RESPONSE          COUNTER_HTTP_SERVER_REQUEST           COUNTER_HTTP_SERVER_RESPONSE          COUNTER_NET_SERVER_CONNECTION
COUNTER_NET_SERVER_CONNECTION_CLOSE   DTRACE_HTTP_CLIENT_REQUEST            DTRACE_HTTP_CLIENT_RESPONSE           DTRACE_HTTP_SERVER_REQUEST
DTRACE_HTTP_SERVER_RESPONSE           DTRACE_NET_SERVER_CONNECTION          DTRACE_NET_STREAM_END                 DataView
Float32Array                          Float64Array                          GLOBAL                                Int16Array
Int32Array                            Int8Array                             Intl                                  Map
Promise                               Proxy                                 Reflect                               Set
SharedArrayBuffer                     Symbol                                Uint16Array                           Uint32Array
Uint8Array                            Uint8ClampedArray                     WeakMap                               WeakSet
WebAssembly                           _                                     a                                     assert
async_hooks                           b                                     buffer                                c
child_process                         clearImmediate                        clearInterval                         clearTimeout
cluster                               console                               crypto                                dgram
dns                                   domain                                escape                                events
fs                                    global                                http                                  http2
https                                 module                                net                                   os
path                                  perf_hooks                            process                               punycode
querystring                           readline                              repl                                  require
root                                  setImmediate                          setInterval                           setTimeout
stream                                string_decoder                        tls                                   tty
unescape                              url                                   util                                  v8
vm                                    zlib

__defineGetter__                      __defineSetter__                      __lookupGetter__                      __lookupSetter__
__proto__                             constructor                           hasOwnProperty                        isPrototypeOf
propertyIsEnumerable                  toLocaleString                        toString                              valueOf
```

REPL Commands

.help – list of all commands.

```
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the repl
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file
>
```

REPL Commands

.editor - Enter editor mode

```
C:\Users\Anna.DESKTOP-RLP7NAJ>node
> .edit
Invalid REPL keyword
> .editor
// Entering editor mode (^D to finish, ^C to cancel)
let x=1;
let y=++x;
console.log("x",x);
console.log("y",y);
```

REPL Commands

- <ctrl>-C - cancel command.
- <ctrl>-D - finish command.

After pressing control + D

```
C:\Users\Anna.DESKTOP-RLP7NAJ>node
> .edit
Invalid REPL keyword
> .editor
// Entering editor mode (^D to finish, ^C to cancel)
let x=1;
let y=++x;
console.log("x",x);
console.log("y",y);
x 2
y 2
undefined
>
```

Stopping REPL

use **ctrl-c twice** to come out of Node.js REPL.

```
C:\Users\Anna.DESKTOP-RLP7NAJ>node
> .edit
Invalid REPL keyword
> .editor
// Entering editor mode (^D to finish, ^C to cancel)
let x=1;
let y=++x;
console.log("x",x);
console.log("y",y);
x 2
y 2
undefined
>
```

Node.js – Event loop

Node.js - Event Loop

Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**. Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

Event-Driven Programming

Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies. As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.

The observer pattern

Pattern (observer): defines an object (called subject), which can notify a set of observers (or listeners), when a change in its state happens.

Observer is an ideal solution for modeling the reactive nature of Node.js, and a perfect complement for callbacks.

The main difference from the callback pattern is that the subject can actually notify multiple observers, while a traditional continuation-passing style callback will usually propagate its result to only one listener, the callback.

Creating an event-emitter

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();

// Bind event and event  handler as follows
eventEmitter.on('eventName', eventHandler);

// Fire an event
eventEmitter.emit('eventName');
```